Rate Limiting en Producción

Token Bucket, Sliding Window y Redis

PuigFlows — Josué Puig

Guía · Nivel Intermedio · Proyectos Reales

Índice general

Versión en Español

Protege tus APIs del abuso y los picos de tráfico con rate limiting. Compara los algoritmos fixed window, sliding window y token bucket, impleméntalos de forma atómica en Redis con Lua, devuelve las cabeceras 429 correctas y evita los errores que invalidan tus límites en silencio. Con código listo para producción.

Qué es el rate limiting y por qué lo necesitas

Toda API que vive en internet recibe, tarde o temprano, más peticiones de las que debería: un cliente con un bucle mal escrito, un scraper agresivo, un ataque de fuerza bruta contra el login o simplemente un pico de tráfico legítimo que tu base de datos no aguanta. El rate limiting es el mecanismo que decide cuántas peticiones acepta tu sistema de un mismo origen en una ventana de tiempo, y qué hacer con las que sobran.

No es solo una medida de seguridad. Un buen límite protege tu infraestructura de la saturación, reparte la capacidad de forma justa entre todos tus usuarios, contiene el coste de servicios externos que pagas por llamada (piensa en la API de un LLM) y te da una palanca para diferenciar planes de pago. Hacerlo mal, en cambio, frustra a usuarios legítimos con bloqueos arbitrarios o deja pasar justo el abuso que pretendías frenar.

Los cuatro algoritmos que conviene conocer

No existe un único «rate limiter». Hay una familia de algoritmos, y cada uno hace un compromiso distinto entre precisión, memoria y tolerancia a las ráfagas. Estos son los que te vas a encontrar en la práctica.

Fixed window (ventana fija)

El más simple: cuenta las peticiones dentro de bloques de tiempo fijos —por ejemplo «100 por minuto», contadas desde el segundo cero de cada minuto— y reinicia el contador cuando el reloj cambia de minuto. Es barato y trivial de entender.

Su talón de Aquiles es el efecto de borde: un cliente puede mandar 100 peticiones en el segundo 59 y otras 100 en el segundo 01 del minuto siguiente, colando 200 peticiones en dos segundos sin violar formalmente el límite. Esa ráfaga en la frontera lo descarta para límites estrictos.

Sliding window log (registro deslizante)

Guarda el timestamp exacto de cada petición. Para decidir si aceptas una nueva, borras los registros más viejos que la ventana (todo lo anterior a «ahora menos 60 segundos») y cuentas lo que queda. Es exacto al milisegundo y elimina por completo el problema del borde.

El precio es la memoria: guardas una entrada por cada petición dentro de la ventana. Con miles de peticiones por segundo y usuario, eso se dispara. En Redis se implementa con un sorted set (ZSET) puntuado por el timestamp.

Sliding window counter (contador deslizante)

Un punto intermedio elegante. Mantienes el contador de la ventana actual y el de la anterior, y aproximas el ritmo ponderando el contador previo por la fracción de la ventana que todavía solapa:

```
peticiones ≈ contador_actual + contador_previo × (1 − fracción_transcurrida)
```

Suaviza el efecto de borde del fixed window usando solo dos enteros, así que la memoria es mínima y la precisión más que suficiente para la mayoría de APIs. Es la opción por defecto recomendada cuando no tienes una razón concreta para elegir otra.

Token bucket (cubo de tokens)

El modelo mental: un cubo con sitio para N tokens que se rellena a ritmo constante (digamos 10 tokens por segundo). Cada petición consume un token; si el cubo está vacío, la petición se rechaza. El cubo se llena hasta su capacidad máxima y ahí se queda.

Su virtud es que permite ráfagas controladas: un cliente que ha estado inactivo acumula tokens y puede gastarlos de golpe, hasta el tamaño del cubo, siempre que su ritmo medio se mantenga bajo control. Es el algoritmo favorito de las APIs públicas —lo usan AWS, Stripe y muchas más— precisamente porque modela bien el comportamiento real: una ráfaga corta está bien, el abuso sostenido no.

Su pariente cercano, el leaky bucket (cubo con fuga), procesa las peticiones a ritmo constante, como agua que sale por un agujero. Es útil cuando quieres suavizar el tráfico hacia un sistema aguas abajo, pero penaliza las ráfagas en vez de permitirlas.

GCRA (Generic Cell Rate Algorithm)

Si quieres lo mejor del token bucket pero con un estado mínimo, mira el GCRA. En lugar de guardar tokens, almacena un único valor por cliente: el «tiempo teórico de llegada» (TAT), el instante en que la próxima petición sería admisible al ritmo configurado. Cada petición empuja ese instante hacia adelante según su coste; si llega demasiado pronto respecto al TAT —más allá de la tolerancia a ráfagas— se rechaza. Una sola clave por cliente, ráfagas controladas y precisión continua, sin ventanas que reiniciar.

Es el algoritmo que popularizó el módulo redis-cell (comando CL.THROTTLE) y que Redis incorporó de forma nativa en versiones recientes (Redis 8.8). Si prefieres no mantener tu propio script Lua, es la opción más limpia y probada para producción.

Por qué Redis y por qué Lua

En cuanto tu servicio corre en más de un proceso —y en producción siempre lo hace— el contador no puede vivir en la memoria de cada instancia: cada una llevaría su propia cuenta y el límite real sería N veces mayor del que crees. Necesitas un estado compartido y rápido, y Redis es la elección estándar por su latencia submilisegundo y sus estructuras de datos atómicas.

El detalle crítico es la atomicidad. Un rate limiter hace «lee el contador, decide, escribe el nuevo valor». Si dos peticiones ejecutan ese ciclo a la vez, ambas pueden leer el mismo valor antiguo y colarse: la condición de carrera de manual. La forma correcta de evitarla es meter toda la lógica en un script Lua, que Redis garantiza ejecutar de forma atómica, sin que ninguna otra orden se intercale. Por eso los ejemplos que siguen encierran la decisión completa en Lua en lugar de hacer varias llamadas sueltas desde la aplicación.

Token bucket en Redis con Lua

El script guarda dos campos en un hash de Redis: los tokens disponibles y el instante del último rellenado. En cada petición calcula cuántos tokens se han regenerado desde la última vez, los suma (sin pasar de la capacidad) e intenta consumir uno.

```lua
-- KEYS[1]: clave del cubo para este cliente (p. ej. rl:tb:user:123)
-- ARGV[1]: capacidad maxima del cubo
-- ARGV[2]: tokens que se reponen por segundo
-- ARGV[3]: timestamp actual en segundos (puede ser decimal)
-- ARGV[4]: tokens a consumir en esta peticion (normalmente 1)
local capacity = tonumber(ARGV[1])
local refill = tonumber(ARGV[2])
local now = tonumber(ARGV[3])
local requested = tonumber(ARGV[4])

local data = redis.call('HMGET', KEYS[1], 'tokens', 'ts')
local tokens = tonumber(data[1])
local last = tonumber(data[2])

-- Primera vez que vemos al cliente: el cubo empieza lleno
if tokens == nil then
  tokens = capacity
  last = now
end

-- Reponer tokens segun el tiempo transcurrido, sin pasar de la capacidad
local elapsed = math.max(0, now - last)
tokens = math.min(capacity, tokens + elapsed * refill)

local allowed = tokens >= requested
if allowed then
  tokens = tokens - requested
end

-- Persistir estado y expirar la clave para no acumular cubos muertos
redis.call('HSET', KEYS[1], 'tokens', tokens, 'ts', now)
redis.call('EXPIRE', KEYS[1], math.ceil(capacity / refill) * 2)

-- Devuelve: permitido (1/0) y tokens restantes redondeados hacia abajo
return { allowed and 1 or 0, math.floor(tokens) }
```

Tres detalles que marcan la diferencia: la reposición se calcula en función del tiempo transcurrido en lugar de con un temporizador, así que no necesitas ningún proceso de fondo; el `EXPIRE` evita que se acumulen cubos de clientes que ya no volverán; y se devuelven los tokens restantes para poder rellenar la cabecera `RateLimit-Remaining`.

El envoltorio en Python con redis-py registra el script una sola vez y lo invoca por su SHA:

```python
import time
import redis

r = redis.Redis(host="localhost", port=6379, decode_responses=True)

# El script se registra una vez; Redis lo cachea por su SHA y lo reutiliza.
```

```python
_token_bucket = r.register_script(LUA_TOKEN_BUCKET)

def allow_request(client_id, capacity=20, refill_per_sec=10.0, cost=1):
    """Devuelve (permitido, tokens_restantes) para un cliente.

    capacity: rafaga maxima que toleramos de una sola vez.
    refill_per_sec: ritmo sostenido que permitimos a largo plazo.
    """
    allowed, remaining = _token_bucket(
        keys=[f"rl:tb:{client_id}"],
        args=[capacity, refill_per_sec, time.time(), cost],
    )
    return bool(allowed), int(remaining)
```

Sliding window counter en Redis

Si prefieres un límite estricto con memoria mínima, este es el algoritmo. Usa dos claves enteras —la ventana actual y la anterior— y pondera la anterior por lo que todavía solapa. La aplicación calcula a qué ventana pertenece «ahora» y qué fracción de ella ha pasado, y se lo entrega al script:

```lua
-- KEYS[1]: contador de la ventana actual (rl:sw:user:123:<ventana>)
-- KEYS[2]: contador de la ventana anterior
-- ARGV[1]: limite de peticiones por ventana
-- ARGV[2]: duracion de la ventana en segundos
-- ARGV[3]: fraccion ya transcurrida de la ventana actual (0..1)
local limit = tonumber(ARGV[1])
local window = tonumber(ARGV[2])
local elapsed_pct = tonumber(ARGV[3])

local current = tonumber(redis.call('GET', KEYS[1]) or '0')
local previous = tonumber(redis.call('GET', KEYS[2]) or '0')

-- Conteo ponderado: la ventana previa pesa segun lo que aun solapa
local estimated = current + previous * (1 - elapsed_pct)

if estimated >= limit then
  return { 0, 0 } -- rechazado
end

current = redis.call('INCR', KEYS[1])
redis.call('EXPIRE', KEYS[1], window * 2)

local used = current + previous * (1 - elapsed_pct)
return { 1, math.max(0, math.floor(limit - used)) }
```

Frente al sliding window log, aquí no almacenas una entrada por petición: dos contadores bastan, sea cual sea el volumen. La aproximación introduce un error pequeñísimo en los bordes, irrelevante para casi cualquier API real.

Devuelve las cabeceras correctas

Cuando rechaces una petición, hazlo con el código HTTP `429 Too Many Requests`, nunca con un 403 o un 500 genérico: el 429 le dice explícitamente al cliente que vuelva a intentarlo más tarde. Acompáñalo de la cabecera `Retry-After` con los segundos que debe esperar.

Para que los clientes educados se autorregulen antes de chocar con el límite, incluye en todas las respuestas (no solo en las 429) las cabeceras `RateLimit-Limit`, `RateLimit-Remaining` y `RateLimit-Reset`, hoy en proceso de estandarización por el IETF. Las variantes con prefijo `X-` (`X-RateLimit-*`) siguen siendo muy comunes; elige una convención y sé consistente. Un middleware de FastAPI lo resume bien:

```
import math
from fastapi import FastAPI, Request
from fastapi.responses import JSONResponse

app = FastAPI()
CAPACITY, REFILL = 20, 10.0

@app.middleware("http")
async def rate_limit(request: Request, call_next):
    # Limita por identidad real cuando puedas; la IP es el ultimo recurso.
    client_id = request.headers.get("X-API-Key") or request.client.host
    allowed, remaining = allow_request(client_id, CAPACITY, REFILL)

    if not allowed:
        retry_after = math.ceil(1 / REFILL)
        return JSONResponse(
            status_code=429,
            content={"detail": "Demasiadas peticiones. Reduce el ritmo."},
            headers={
                "Retry-After": str(retry_after),
                "RateLimit-Limit": str(CAPACITY),
                "RateLimit-Remaining": "0",
            },
        )

    response = await call_next(request)
    response.headers["RateLimit-Limit"] = str(CAPACITY)
    response.headers["RateLimit-Remaining"] = str(remaining)
    return response
```

Distribuido: ¿fail-open o fail-closed?

Con varias instancias detrás de un balanceador, Redis centraliza el estado y todas comparten el mismo contador. La pregunta incómoda llega sola: ¿qué haces si Redis no responde? Tienes dos opciones. Fail-open (dejar pasar la petición) prioriza la disponibilidad: tu API sigue funcionando, aunque sin protección durante el incidente. Fail-closed (rechazar) prioriza la protección a costa de tirar el servicio.

Para la mayoría de APIs públicas, fail-open con una alerta inmediata al equipo es la decisión correcta: un rato sin rate limiting es menos grave que una caída total. Para endpoints muy sensibles —login,

pagos— puede merecer la pena fail-closed. Lo importante es decidirlo a conciencia, no por accidente.

Cuando creces de verdad, una sola instancia de Redis se queda corta y pasas a Redis Cluster, que reparte las claves en 16.384 slots entre varios nodos. Si tu script toca varias claves a la vez, fuérzalas al mismo nodo con hash tags, por ejemplo `rl:{user:123}:actual` y `rl:{user:123}:previa`: lo que va entre llaves es lo único que Redis usa para calcular el slot.

Errores comunes que invalidan tus límites

- **Contar en memoria local con varias réplicas.** Cada proceso lleva su cuenta y el límite efectivo se multiplica por el número de instancias. Centraliza siempre el estado.
- **Hacer GET y luego SET por separado.** Abres una condición de carrera. Mete la lógica en un script Lua atómico o apóyate en `INCR`.
- **Limitar por IP sin pensarlo.** Detrás de un NAT corporativo o un proxy móvil hay miles de usuarios con la misma IP. Limita por user id o API key cuando puedas, y deja la IP como último recurso.
- **Olvidar el TTL de las claves.** Sin `EXPIRE`, Redis acumula claves de clientes que no volverán y la memoria crece sin freno.
- **Usar el reloj de la aplicación en un sistema distribuido.** Relojes desincronizados entre instancias corrompen las ventanas. Apóyate en el tiempo del propio Redis (comando `TIME`) cuando la precisión importe.
- **Aplicar el mismo límite a todo.** Un endpoint de búsqueda caro y un health check no deberían compartir presupuesto.

Buenas prácticas, en una frase

Elige token bucket si quieres permitir ráfagas y sliding window counter si quieres un límite estricto con poca memoria; ejecuta la decisión de forma atómica en Redis con Lua; limita por identidad de usuario antes que por IP; responde con 429, Retry-After y cabeceras RateLimit-*; decide a conciencia entre fail-open y fail-closed; pon TTL a todas las claves; y monitoriza los rechazos para distinguir un ataque de un límite mal calibrado.

English Version

Protect your APIs from abuse and traffic spikes with rate limiting. Compare the fixed window, sliding window, and token bucket algorithms, implement them atomically in Redis with Lua, return the correct 429 headers, and avoid the mistakes that silently void your limits. With production-ready code.

What rate limiting is, and why you need it

Every API exposed to the internet eventually receives more requests than it should: a client stuck in a bad loop, an aggressive scraper, a brute-force attack against your login, or simply a spike of legitimate traffic your database cannot absorb. Rate limiting is the mechanism that decides how many requests your system accepts from a single origin within a time window, and what to do with the ones that exceed it.

It is more than a security measure. A good limit shields your infrastructure from overload, shares capacity fairly across all your users, caps the cost of pay-per-call upstream services (think of an LLM API), and gives you a lever to differentiate paid tiers. Done badly, it frustrates legitimate users with arbitrary blocks or lets through exactly the abuse it was meant to stop.

The four algorithms worth knowing

There is no single "rate limiter." It is a family of algorithms, each making a different trade-off between accuracy, memory, and burst tolerance. These are the ones you will meet in practice.

Fixed window

The simplest: count requests inside fixed time blocks —say "100 per minute," counted from second zero of each minute— and reset the counter when the clock ticks over. It is cheap and trivial to understand.

Its weak spot is the boundary effect: a client can fire 100 requests at second 59 and another 100 at second 01 of the next minute, sneaking 200 requests through in two seconds without formally breaking the limit. That edge burst rules it out for strict limits.

Sliding window log

Store the exact timestamp of every request. To decide on a new one, drop the entries older than the window (everything before "now minus 60 seconds") and count what remains. It is accurate to the millisecond and removes the boundary problem entirely.

The cost is memory: you keep one entry per request inside the window. At thousands of requests per second per user, that explodes. In Redis it is implemented with a sorted set (ZSET) scored by timestamp.

Sliding window counter

An elegant middle ground. You keep the counter for the current window and the previous one, and approximate the rate by weighting the previous counter by the fraction of the window that still overlaps:

```
requests ≈ current_count + previous_count × (1 − elapsed_fraction)
```

It smooths out the boundary effect of the fixed window using just two integers, so memory is minimal and accuracy is more than enough for most APIs. It is the recommended default when you have no specific reason to pick another.

Token bucket

The mental model: a bucket holding up to N tokens that refills at a constant rate (say 10 tokens per second). Each request consumes one token; if the bucket is empty, the request is rejected. The bucket fills up to its maximum capacity and stops there.

Its strength is that it allows controlled bursts: a client that has been idle accumulates tokens and can spend them all at once, up to the bucket size, as long as its average rate stays in check. It is the favourite of public APIs —AWS, Stripe, and many others use it— precisely because it models real behaviour well: a short burst is fine, sustained abuse is not.

Its close relative, the leaky bucket, processes requests at a constant rate, like water draining through a hole. It is useful when you want to smooth traffic toward a downstream system, but it penalises bursts instead of allowing them.

GCRA (Generic Cell Rate Algorithm)

If you want the best of the token bucket but with minimal state, look at GCRA. Instead of storing tokens, it keeps a single value per client: the "theoretical arrival time" (TAT), the moment at which the next request would be admissible at the configured rate. Each request pushes that moment forward by its cost; if it arrives too early relative to the TAT —beyond the burst tolerance— it is rejected. One key per client, controlled bursts, and continuous accuracy, with no windows to reset.

It is the algorithm popularised by the redis-cell module (the CL.THROTTLE command) and that Redis added natively in recent versions (Redis 8.8). If you would rather not maintain your own Lua script, it is the cleanest, battle-tested option for production.

Why Redis, and why Lua

As soon as your service runs in more than one process —and in production it always does— the counter cannot live in the memory of each instance: each would keep its own count and the real limit would be N times higher than you think. You need shared, fast state, and Redis is the standard choice for its sub-millisecond latency and atomic data structures.

The critical detail is atomicity. A rate limiter does "read the counter, decide, write the new value." If two requests run that cycle at the same time, both can read the same stale value and slip through: the textbook race condition. The right way to avoid it is to put the whole decision inside a Lua script, which Redis guarantees to run atomically, with no other command interleaved. That is why the examples below wrap the entire decision in Lua instead of making several separate calls from the application.

Token bucket in Redis with Lua

The script keeps two fields in a Redis hash: the available tokens and the timestamp of the last refill. On each request it computes how many tokens have regenerated since last time, adds them (capped at capacity), and tries to consume one.

```lua
-- KEYS[1]: bucket key for this client (e.g. rl:tb:user:123)
-- ARGV[1]: maximum bucket capacity
-- ARGV[2]: tokens refilled per second
-- ARGV[3]: current timestamp in seconds (may be fractional)
-- ARGV[4]: tokens to consume on this request (usually 1)
local capacity = tonumber(ARGV[1])
local refill = tonumber(ARGV[2])
local now = tonumber(ARGV[3])
local requested = tonumber(ARGV[4])

local data = redis.call('HMGET', KEYS[1], 'tokens', 'ts')
local tokens = tonumber(data[1])
local last = tonumber(data[2])

-- First time we see this client: the bucket starts full
if tokens == nil then
  tokens = capacity
  last = now
end

-- Refill tokens based on elapsed time, never above capacity
local elapsed = math.max(0, now - last)
tokens = math.min(capacity, tokens + elapsed * refill)

local allowed = tokens >= requested
if allowed then
  tokens = tokens - requested
end

-- Persist state and expire the key so dead buckets do not pile up
redis.call('HSET', KEYS[1], 'tokens', tokens, 'ts', now)
redis.call('EXPIRE', KEYS[1], math.ceil(capacity / refill) * 2)

-- Returns: allowed (1/0) and remaining tokens, floored
return { allowed and 1 or 0, math.floor(tokens) }
```

Three details make the difference: the refill is computed from elapsed time rather than from a timer, so you need no background process; the `EXPIRE` stops buckets for clients who never return from piling up; and the remaining tokens are returned so you can fill the `RateLimit-Remaining` header.

The Python wrapper with redis-py registers the script once and calls it by its SHA:

```python
import time
import redis

r = redis.Redis(host="localhost", port=6379, decode_responses=True)

# The script is registered once; Redis caches it by SHA and reuses it.
_token_bucket = r.register_script(LUA_TOKEN_BUCKET)
```

```
def allow_request(client_id, capacity=20, refill_per_sec=10.0, cost=1):
    """Return (allowed, remaining_tokens) for a client.

    capacity: largest burst we tolerate in one go.
    refill_per_sec: sustained rate we allow over the long run.
    """
    allowed, remaining = _token_bucket(
        keys=[f"rl:tb:{client_id}"],
        args=[capacity, refill_per_sec, time.time(), cost],
    )
    return bool(allowed), int(remaining)
```

Sliding window counter in Redis

If you prefer a strict limit with minimal memory, this is the algorithm. It uses two integer keys —the current window and the previous one— and weights the previous one by what still overlaps. The application works out which window "now" falls into and how much of it has elapsed, and hands that to the script:

```
-- KEYS[1]: current window counter (rl:sw:user:123:WINDOW)
-- KEYS[2]: previous window counter
-- ARGV[1]: request limit per window
-- ARGV[2]: window length in seconds
-- ARGV[3]: fraction already elapsed of the current window (0..1)
local limit = tonumber(ARGV[1])
local window = tonumber(ARGV[2])
local elapsed_pct = tonumber(ARGV[3])

local current = tonumber(redis.call('GET', KEYS[1]) or '0')
local previous = tonumber(redis.call('GET', KEYS[2]) or '0')

-- Weighted count: the previous window counts only for the part that still overlaps
local estimated = current + previous * (1 - elapsed_pct)

if estimated >= limit then
  return { 0, 0 } -- rejected
end

current = redis.call('INCR', KEYS[1])
redis.call('EXPIRE', KEYS[1], window * 2)

local used = current + previous * (1 - elapsed_pct)
return { 1, math.max(0, math.floor(limit - used)) }
```

Unlike the sliding window log, you store no per-request entry here: two counters are enough, whatever the volume. The approximation introduces a tiny error at the boundaries, irrelevant for almost any real API.

Return the right headers

When you reject a request, do it with HTTP `429 Too Many Requests`, never a 403 or a generic 500: the 429 tells the client explicitly to try again later. Pair it with the `Retry-After` header carrying the number of seconds to wait.

So that well-behaved clients self-throttle before hitting the wall, include on every response (not only on 429s) the `RateLimit-Limit`, `RateLimit-Remaining`, and `RateLimit-Reset` headers, currently being standardised by the IETF. The `X-` prefixed variants (`X-RateLimit-*`) are still very common; pick one convention and stay consistent. A FastAPI middleware sums it up:

```
import math
from fastapi import FastAPI, Request
from fastapi.responses import JSONResponse

app = FastAPI()
CAPACITY, REFILL = 20, 10.0

@app.middleware("http")
async def rate_limit(request: Request, call_next):
    # Prefer a real identity; fall back to IP only as a last resort.
    client_id = request.headers.get("X-API-Key") or request.client.host
    allowed, remaining = allow_request(client_id, CAPACITY, REFILL)

    if not allowed:
        retry_after = math.ceil(1 / REFILL)
        return JSONResponse(
            status_code=429,
            content={"detail": "Too many requests. Slow down."},
            headers={
                "Retry-After": str(retry_after),
                "RateLimit-Limit": str(CAPACITY),
                "RateLimit-Remaining": "0",
            },
        )

    response = await call_next(request)
    response.headers["RateLimit-Limit"] = str(CAPACITY)
    response.headers["RateLimit-Remaining"] = str(remaining)
    return response
```

Distributed: fail-open or fail-closed?

With several instances behind a load balancer, Redis centralises the state and they all share the same counter. The uncomfortable question follows on its own: what do you do if Redis is unreachable? You have two options. Fail-open (let the request through) prioritises availability: your API keeps working, though unprotected during the incident. Fail-closed (reject) prioritises protection at the cost of taking the service down.

For most public APIs, fail-open with an immediate alert to the team is the right call: a spell without rate limiting is less serious than a full outage. For very sensitive endpoints —login, payments— fail-

closed may be worth it. What matters is choosing deliberately, not by accident.

When you grow for real, a single Redis instance falls short and you move to Redis Cluster, which spreads keys across 16,384 slots over several nodes. If your script touches several keys at once, force them onto the same node with hash tags, e.g. `rl:{user:123}:current` and `rl:{user:123}:previous`: only what sits between the braces is used to compute the slot.

Common mistakes that void your limits

- **Counting in local memory with several replicas.** Each process keeps its own count and the effective limit multiplies by the number of instances. Always centralise the state.
- **Doing GET then SET separately.** You open a race condition. Put the logic in an atomic Lua script or lean on `INCR`.
- **Limiting by IP without thinking.** Behind a corporate NAT or a mobile proxy there are thousands of users on the same IP. Limit by user id or API key when you can, and treat the IP as a last resort.
- **Forgetting key TTLs.** Without `EXPIRE`, Redis hoards keys for clients who never return and memory grows unbounded.
- **Using the application clock in a distributed system.** Clocks drifting between instances corrupt the windows. Lean on the time of Redis itself (the `TIME` command) when precision matters.
- **Applying the same limit to everything.** An expensive search endpoint and a health check should not share a budget.

Best practices, in one sentence

Pick token bucket if you want to allow bursts and sliding window counter if you want a strict limit with little memory; run the decision atomically in Redis with Lua; limit by user identity before IP; respond with 429, Retry-After, and RateLimit-* headers; choose deliberately between fail-open and fail-closed; give every key a TTL; and monitor rejections to tell an attack apart from a badly tuned limit.