← All articles Architecture

Rate Limiting Patterns: Protecting Your APIs Without Blocking Legitimate Traffic

Token bucket, sliding window, adaptive limits. How to implement rate limiting that stops abuse without punishing your users, with examples for Traefik, Nginx, and application-level throttling.

Y Yash Pritwani 24 March 2026 11 min read read

Every production API faces the same threat: a flood of requests that overwhelms your infrastructure, whether from a malfunctioning client, a competitor scraping your data, or a genuine DDoS attack. Rate limiting is your first line of defense — but implementing it poorly means blocking legitimate users while bad actors find workarounds. This guide covers the algorithms, patterns, and real-world configurations that give you precise control over traffic without turning away your best customers.

Why Naive Rate Limiting Fails

The simplest approach — "allow 100 requests per minute, then block" — sounds reasonable until it hits production. A user who sends 100 requests at 12:00:00 and one more at 12:00:01 gets blocked. Meanwhile, a scraper that spaces requests perfectly sails through your limit.

The Core Algorithms

Fixed Window Counter

Divide time into fixed windows and count requests per window. When the counter hits the limit, reject requests until the window resets.

```
import redis
import time

def fixed_window_check(client_id: str, limit: int, window_seconds: int) -> bool:
    r = redis.Redis()
    window_key = f"ratelimit:{client_id}:{int(time.time() // window_seconds)}"
    current = r.incr(window_key)
    if current == 1:
```

```
        r.expire(window_key, window_seconds)
    return current <= limit
```

Sliding Window Counter (Hybrid)

Approximates the sliding log using two fixed windows weighted by elapsed time:

```
def sliding_window_counter_check(client_id: str, limit: int, window_seconds: int) -> bool:
    r = redis.Redis()
    now = time.time()
    current_window = int(now // window_seconds)
    previous_window = current_window - 1
    weight = 1 - ((now % window_seconds) / window_seconds)
    prev_count = int(r.get(f"ratelimit:swc:{client_id}:{previous_window}") or 0)
    curr_count = int(r.get(f"ratelimit:swc:{client_id}:{current_window}") or 0)
    estimated_count = (prev_count * weight) + curr_count
    if estimated_count >= limit:
        return False
    pipe = r.pipeline()
    pipe.incr(f"ratelimit:swc:{client_id}:{current_window}")
    pipe.expire(f"ratelimit:swc:{client_id}:{current_window}", window_seconds * 2)
    pipe.execute()
    return True
```

Token Bucket

A bucket fills with tokens at a constant rate. Each request consumes one token. Empty bucket means rejected request. Token bucket elegantly handles burst traffic.

```
def token_bucket_check(client_id: str, capacity: int, refill_rate: float) -> bool:
    r = redis.Redis()
    key = f"ratelimit:tb:{client_id}"
    now = time.time()
    lua_script = '''
    local key = KEYS[1]
    local capacity = tonumber(ARGV[1])
    local refill_rate = tonumber(ARGV[2])
    local now = tonumber(ARGV[3])
    local data = redis.call("HMGET", key, "tokens", "last_refill")
    local tokens = tonumber(data[1]) or capacity
    local last_refill = tonumber(data[2]) or now
    local elapsed = now - last_refill
    local new_tokens = math.min(capacity, tokens + (elapsed * refill_rate))
    if new_tokens < 1 then return 0 end
```

```
    redis.call("HMSET", key, "tokens", new_tokens - 1, "last_refill", now)
    redis.call("EXPIRE", key, math.ceil(capacity / refill_rate) + 1)
    return 1
    '''
    result = r.eval(lua_script, 1, key, capacity, refill_rate, now)
    return bool(result)
```

Algorithm Comparison

Algorithm	Burst Handling	Memory Usage	Accuracy	Best For
Fixed Window	Poor	Very Low	Low	Simple internal APIs
Sliding Window Log	Excellent	High	Exact	Low-traffic, high-precision
Sliding Window Counter	Good	Very Low	~99%	High-traffic APIs
Token Bucket	Excellent	Low	High	Public APIs with bursty clients
Leaky Bucket	Smooths bursts	Medium	High	Protecting downstream services

Per-User vs Per-IP Rate Limiting

IP-based limiting is the easiest to implement but punishes users behind NAT or corporate proxies. **Per-user limiting** requires authentication but gives precise per-client control.

A production system typically layers both:

```
from fastapi import Request, HTTPException

def rate_limit(requests_per_minute: int, per: str = "user"):
    def decorator(func):
        async def wrapper(request: Request, *args, **kwargs):
            if per == "user":
                client_id = request.state.user_id
            else:
                client_id = request.client.host
            allowed = token_bucket_check(
                client_id=f"{per}:{client_id}",
                capacity=requests_per_minute,
                refill_rate=requests_per_minute / 60.0
            )
            if not allowed:
                raise HTTPException(status_code=429, detail="Rate limit exceeded",
                    headers={"Retry-After": "60"})
            return await func(request, *args, **kwargs)
        return wrapper
    return decorator
```

Traefik Middleware Configuration


```yaml
http:
  middlewares:
    api-ratelimit:
      rateLimit:
        average: 100
        period: 1m
        burst: 50
        sourceCriterion:
          ipStrategy:
            depth: 1

  routers:
    api-router:
      rule: "Host(`api.example.com`)"
      middlewares:
        - api-ratelimit
      service: api-service
```

You might also like

→ eBPF Beyond Security: Networking, Observability, and Performance in One Technology
12 min read read

→ Edge AI Inference: Why the Cloud Is Too Slow and How to Deploy Models at the Edge
11 min read

→ Proxmox Clustering: High Availability for Your Self-Hosted Infrastructure
12 min read read

Or via Docker labels:

```yaml
services:
  api:
    image: your-api:latest
    labels:
      - "traefik.http.middlewares.api-ratelimit.ratelimit.average=100"
      - "traefik.http.middlewares.api-ratelimit.ratelimit.period=1m"
      - "traefik.http.middlewares.api-ratelimit.ratelimit.burst=20"
```

Nginx Configuration

```nginx
http {
    limit_req_zone $binary_remote_addr zone=per_ip:10m rate=10r/s;
    limit_req_zone $http_x_api_key zone=per_key:10m rate=100r/m;
    limit_req_status 429;

    server {
```

```
        location /api/ {
            limit_req zone=per_ip burst=20 nodelay;
            limit_req zone=per_key burst=50;
            proxy_pass http://backend;
        }
        error_page 429 @rate_limited;
        location @rate_limited {
            add_header Retry-After 60 always;
            return 429 '{"error": "Rate limit exceeded"}';
        }
    }
}
```

Distributed Rate Limiting with Redis

Single-node rate limiting breaks when you scale horizontally. Redis provides a shared, atomic counter store. The token bucket Lua script handles this — every instance hits the same Redis key, and Lua ensures atomicity.

For high-availability:

```
from redis.sentinel import Sentinel
sentinel = Sentinel([("sentinel1", 26379), ("sentinel2", 26379)])
master = sentinel.master_for("mymaster", socket_timeout=0.1)
```

Adaptive Rate Limiting

Static limits are a blunt instrument. Adjust limits based on current system load:


```
import psutil

class AdaptiveLimits:
    def __init__(self, base_limit=100):
        self.base_limit = base_limit

    def get_current_limit(self) -> int:
        load = max(psutil.cpu_percent(interval=0.1), psutil.virtual_memory().percent) / 100
        if load >= 0.95:
            return max(10, int(self.base_limit * 0.3))
        elif load >= 0.80:
```

```
            return max(10, int(self.base_limit * 0.6))
        return self.base_limit
```

Communicating Limits to Clients

Always return standardized headers:

```
X-RateLimit-Limit: 100
X-RateLimit-Remaining: 42
X-RateLimit-Reset: 1711324800
Retry-After: 60
```

The `Retry-After` header is critical — without it, clients guess when to retry, creating thundering-herd problems.

Common Pitfalls

- **Not rate limiting by endpoint**: A bulk export should have tighter limits than a health check
- **Ignoring auth vs unauth traffic**: Anonymous requests deserve stricter limits
- **Forgetting Redis failure handling**: Decide whether to fail open or closed
- **Not testing limits**: Write integration tests that actually hit the limits

Summary

Rate limiting is not a single setting but a layered strategy: network-level blocking via Traefik or Nginx catches bulk abuse cheaply, while application-level logic handles nuanced per-user policies. Token bucket suits bursty legitimate clients, sliding window counter suits high-precision enforcement, and adaptive limits protect infrastructure during unexpected load spikes. Start with Traefik middleware for the 80% case, add Redis-backed per-user limits for authenticated APIs, and layer adaptive limiting for production resilience.

#rate-limiting #api-design #traefik #nginx #throttling #distributed-systems #security

TE TechSaaS

Need help with architecture?

TechSaaS provides expert consulting and managed services for cloud infrastructure, DevOps, and AI/ML operations.

Get a Free Consultation | WhatsApp Us

We Will Build You a Demo Site — For Free

Like it? Pay us. Do not like it? Walk away, zero complaints. You will spend way less than hiring developers or any agency.

47+ companies trusted us | **99.99%** uptime | **< 48hr** response

your@email.com

+91 98765 43210 (optional)

Get My Free Demo

No spam. No contracts. Just a free demo.

Related Articles

Architecture
WebAssembly on the Server: Running Wasm Workloads...
11 min read

Architecture
Durable Execution: Why Your Workflows Should...
7 min read

Security
Zero-Trust Security Architecture for Cloud-...
13 min

TE TechSaaS

Stay in the Loop

Get product updates, engineering blog posts, and tech insights. No spam, ever.

you@company.com

Subscribe

TE TechSaaS

Full-stack product studio delivering AI-powered platforms, SaaS products, and enterprise solutions across HR-Tech, Ed-Te...

PRODUCTS

- Skillety (AI Hiring)
- Entrance (3D Learning)
- OpenClaw (AI Gateway)
- PADC (Cloud Platform)
- View All Products

SERVICES

- AI & ML Solutions
- Full-Stack Development
- Cloud & DevOps
- Product Consulting

COMPANY

- About Us
- Careers
- Blog
- Contact

LEGAL

- Privacy Policy
- Terms of Service
- Cookie Policy
- Shipping Policy
- Refund Policy
- Payment Terms

Privacy Terms Cookies Shipping Refund Payment Terms