Lesson 8 -> Rate Limiters

1️⃣ What is a Rate Limiter & Why It Exists

What is a Rate Limiter?

A **rate limiter controls how many requests a client can make to a system within a given time window**.

Example rules:

- 100 requests per minute per user
- 10 inference calls per second per API key
- 1,000 requests/day for free tier

Why Do We Need Rate Limiters?

1. Protect System Stability

Without rate limiting:

- One buggy client
- One malicious actor
- One traffic spike

👉 Can bring your entire system down.

---

2. Cost Control (Critical for ML)

ML inference is expensive:

- GPU usage
- Model loading
- Latency amplification

Rate limiting prevents:

- Cost explosion
- GPU starvation

---

3. Fair Usage

Ensures:

- One customer doesn’t starve others
- Tier-based access (free vs paid)

---

4. Security

Mitigates:

- DDoS attacks
- Brute-force attempts
- Credential stuffing

Where Rate Limiters Sit

Most commonly:
Client
→ API Gateway
→ Rate Limiter
→ Services / ML models

2️⃣ Rate Limiting Algorithms (Most Important Section)

There is **no single best algorithm**.
Each trades **accuracy, memory, fairness, and performance**.

2.1 Fixed Window Counter

How it works

- Define a window (e.g. 1 minute)
- Count requests in that window
- Reset count when window ends

Example:

- Max 100 req/min
- Counter resets at 12:01, 12:02, etc.

Pros

- Simple
- Easy to implement
- Low memory

Cons

❌ Burst problem
A user can send:

- 100 requests at 12:00:59
- 100 requests at 12:01:01

👉 200 requests in 2 seconds

2.2 Sliding Window Log

How it works

- Store timestamp of every request
- Allow only N requests in last T seconds

Pros

- Very accurate
- Smooth traffic control

Cons

❌ High memory usage
❌ Expensive at scale

Rarely used directly in large systems.

2.3 Sliding Window Counter (Hybrid)

How it works

- Divide time into smaller buckets
- Approximate sliding window using counters

Pros

- Good accuracy
- Less memory than logs

Cons

- Slight approximation errors
- More complex

Used in **real-world gateways**.

2.4 Token Bucket (Most Popular)

How it works

- Tokens refill at fixed rate
- Each request consumes a token
- If no tokens → request rejected

Pros

✅ Allows bursts
✅ Smooth rate over time
✅ Simple math

Cons

- Slight complexity
- Needs centralized state in distributed systems

Widely used in:

- API gateways
- ML inference throttling

2.5 Leaky Bucket

How it works

- Requests enter a queue
- Requests processed at constant rate

Pros

- Very smooth output traffic

Cons

❌ Bursty requests are dropped
❌ Less flexible than token bucket

Algorithm Comparison

Algorithm	Burst Friendly	Accuracy	Complexity
Fixed Window	❌	Low	Low
Sliding Log	❌	Very High	High
Sliding Counter	⚠️	High	Medium
Token Bucket	✅	High	Medium
Leaky Bucket	❌	Medium	Medium

3️⃣ Rate Limiters in Distributed Environments

This is where **real complexity starts**.

Problem:

In microservices:

- Multiple API gateway instances
- Multiple servers
- Autoscaling pods

Each instance has its own memory.

👉 **Local counters won't work**

Approaches

3.1 Centralized Rate Limiter

All instances talk to:

- Redis
- Memcached
- DynamoDB

Pros:

- Strong consistency
- Simple logic

Cons:
❌ Redis becomes hot path
❌ Network latency

3.2 Distributed (Eventually Consistent)

Each node:

- Maintains local counters
- Syncs periodically

Pros:

- Faster
- More scalable

Cons:
❌ Slight inaccuracies

Used when:

- Small violations acceptable
- High throughput systems

3.3 Hybrid (Most Real Systems)

- Local pre-check
- Global Redis enforcement

Example:

- Reject obvious overload locally
- Enforce strict limits globally

4️⃣ Caching in Rate Limiters

Caching is used to:

- Reduce DB hits

- Reduce Redis load
- Improve latency

What is cached?

- User tier (free/premium)
- Rate limit rules
- Temporary counters

Where caching happens?

1. In-memory (per node)
2. Redis (shared cache)
3. CDN / Edge (Cloudflare, Akamai)

Cache Tradeoffs

- TTL must be small
- Stale cache = inaccurate limits
- Must tolerate slight inconsistencies

5️⃣ Why Redis is Used for Rate Limiting

Redis is almost the **industry default**.

Why?

- Extremely fast (in-memory)
- Atomic operations
- Supports TTL
- Distributed-friendly

Typical Redis Operations

- INCR

- EXPIRE
- LUA scripts for atomic logic

Redis + Token Bucket

Redis stores:

- Token count
- Last refill timestamp

Used by:

- Kong
- Envoy
- AWS API Gateway (internally)

6️⃣ High-Level Design of Rate Limiters in Microservices

Single Rate Limiter

```
Client
 → API Gateway
   → Rate Limiter
     → Services
```

Multiple Rate Limiters (Real World)

You often need **multiple layers**:

Client
→ Edge Rate Limiter (IP based)
→ API Gateway (User based)
→ Service Rate Limiter
→ ML Model Rate Limiter

Why Multiple?

- Edge: stop attacks early
- Gateway: protect APIs
- Service: protect backend
- ML: protect GPU

Per-Dimension Rate Limiting

You can limit by:

- IP
- User ID
- API key
- Model version
- Customer tier

7️⃣ Deep-Dive Scenarios (Very Important)

Scenario 1: ML Inference Overload

Problem:

- Sudden spike in recommendations

Solution:

- Token bucket at gateway
- Separate bucket per model
- Hard GPU limits

Scenario 2: One Customer Abusing System

Solution:

- Per-customer Redis keys
- Tier-based limits
- Dynamic throttling

Scenario 3: Distributed API Gateway Autoscaling

Problem:

- New pods start with empty counters

Solution:

- Central Redis
- Warm-up logic
- Consistent hashing

Scenario 4: Graceful Degradation

When limit exceeded:

- Return cached response
- Return simpler model result
- Queue async processing

Scenario 5: Multi-Region Systems

- Local region rate limiting
- Global quota enforcement
- Eventual consistency accepted