API Rate Limiting | Creating our own Rate Limiter

What is Rate Limiting?

Rate Limiting means controlling how many requests a user (or system) can send in a given time.

Example:

A system allows 100 requests per minute per user.
If someone sends more than that, the extra requests are blocked or delayed.

1.Why do we use Rate Limiting? (What problem it solves)

Rate Limiting protects and stabilizes the system.

1. Prevents System Overload

If too many requests come at once, the server can crash.
Rate limiting keeps traffic under control so the system stays fast and stable.

2. Stops Abuse and DoS Attacks

Hackers or bots can send millions of requests to slow down or break the system.

Rate limiting blocks such malicious traffic.

3. Ensures Fair Usage

If one user sends too many requests, others will suffer.

Rate limiting makes sure everyone gets a fair share of resources.

4. Protects Expensive Resources

APIs, databases, or cloud services cost money.
Rate limiting avoids unnecessary or repeated calls.

5. Improves Performance & Reliability

Controlled traffic =
Faster response
Less downtime
More stable system

Simple Real-World Example

Instagram API

Instagram may allow:

500 requests per hour per user

If you exceed this:

Your requests are rejected
This protects Instagram servers from overload and abuse.

Rate Limiting controls how fast requests come, to keep the system safe, stable, and fair.

Where can we apply Rate Limiting

1. Client-Side Rate Limiting

What is it?

Limiting requests on the client (browser or app) before sending to the server.

Why use it?

- Reduces unnecessary traffic
- Saves user's bandwidth
- Improves server performance

Problem

- Client can be modified by the user
- Not secure against bots or hackers
- Lost the control.

So client-side is not reliable for security.

2. Middleware / Gateway Rate Limiting

What is it?

Limiting requests at an API Gateway, Firewall, or Load Balancer.

Why use it?

- Stops abuse before reaching the main server
- Protects backend services
- Centralized control for all APIs

Example

AWS API Gateway throttling, NGINX rate limiting, Cloudflare WAF.

Very efficient and commonly used.

3. Server-Side Rate Limiting

What is it?

Limiting requests inside the application server.

Why use it?

- Full control over business logic
- Can limit based on user roles (free vs premium)
- More flexible

Problem

- Requests already reached the server (waste of resources)
- Less efficient for blocking large attacks

Layer	Security	Efficiency	Flexibility	Use Case
Client Side	Low	Good	Low	UI protection
Middleware/Gateway	High	High	Medium	API protection
Server Side	Medium	Low	High	Business logic limits

Best approach: Apply Rate Limiting at Middleware/Gateway + Server Side

Why?

1.Middleware blocks malicious traffic early
2.Server side handles user-specific or business rules
3.Client side alone is not secure

We apply rate limiting at middleware (API gateway/WAF) for protection and at server-side for logic; middleware is the most efficient and secure.

Use Cases of Rate Limiting

1. API Rate Limiting

APIs use rate limiting to control how many requests a user can send.
It helps to:

- Give fair access to all users
- Stop abuse (like bots sending too many requests)

2. Web Server Rate Limiting

Web servers use rate limiting to:

- Protect against DoS attacks
- Avoid server overload when traffic is too high

3. Database Rate Limiting

Rate limiting is applied to database queries to:

- Protect database performance
- Prevent too many queries from one user

Example:

An e-commerce website limits how many times a user can request product data so the database doesn't crash.

4. Login Rate Limiting

Login systems use rate limiting to:

- Stop brute-force attacks
- Prevent password guessing

They limit:

- Number of login attempts per user or per IP

So attackers can't try unlimited passwords.

Types of Rate Limiting

1. IP-based Rate Limiting

This type limits how many requests come from one IP address in a fixed time.

Example:

Only 10 requests per minute per IP.

It is used to:

- Stop bots
- Prevent DoS (Denial of Service) attacks
- Protect the system from too much traffic

Advantages

- Easy to implement at network or app level
- Helps stop request flooding from one IP
- Good for basic protection

Limitations

- Attackers can bypass using VPN, proxy, or botnets
- If many users share one IP (like office or WiFi),
  a real user can be blocked by mistake

Real-World Example

An online retailer sets a limit of 10 requests per minute per IP.
This stops bots from scraping product data,
while normal users can still browse and shop smoothly.

2. User-Based Rate Limiting

What it is

This type limits how many requests one logged-in user can make in a given time.

Instead of checking the IP, the system checks the User ID / Account.

Why we use it

- Ensures fair usage for every user
- Stops a single user from abusing the system
- Useful when users are logged in
- Better than IP-based limits because each user gets their own quota

How it works

When a user sends a request:

1. System identifies the user ID
2. It checks how many requests that user made in that time window
3. If limit is exceeded → request is blocked or delayed

Example

An app allows:

- **100 requests per hour per user**

So:

- **User A → can make 100 requests**
- **User B → can also make 100 requests**
- **Each user is treated independently**

Advantages

- **Fair for all users**
- **Not affected by shared IPs**
- **Prevents account-level abuse**

Limitations

- **Requires users to be logged in**
- **Attackers can create multiple fake accounts to bypass limits**

3. API Key-Based Rate Limiting

What it is

This type limits requests based on an API key.

Every developer or application gets a unique API key to access your API.

Why we use it

- **Controls how much third-party apps use your API**
- **Prevents API abuse or overuse**
- **Helps you monitor and bill usage**
- **Common in public APIs (Google, Twitter, Stripe)**

How it works

When a request comes:

1. **System checks the API key**
2. **Counts how many requests that key has made**
3. **If it exceeds the limit → request is rejected**

Example

Your API allows:

- **1000 requests per day per API key**

So:

- App 1 (Key A) → 1000 requests/day
- App 2 (Key B) → 1000 requests/day
- Each key has its own separate limit

Advantages

- Works for external apps and developers
- Easy to track usage per application
- Helps in API monetization

Limitations

- If API key is leaked → attacker can use it
- Attackers can create multiple API keys to bypass limit

Real-World Example

Google AI Studio API Key

Imagine:

You are using Gemini API from Google AI Studio.

Google gives you 1 API key.

Your Limit

Google says:

You can send 60 requests per minute.

What Happens

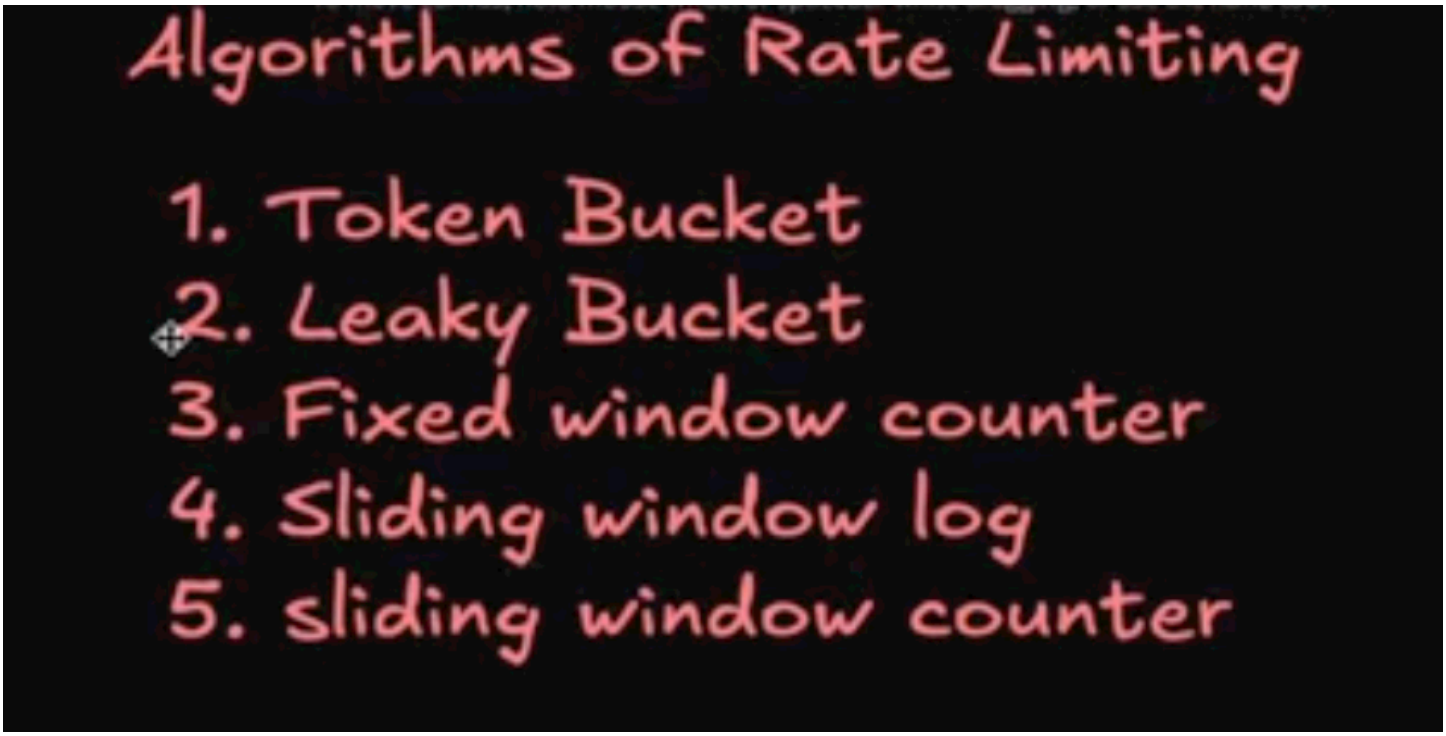
- If you send 50 requests → Everything works fine
- If you send 70 requests →
  Google will block the extra 10 requests
  and show an error: "Rate limit exceeded"

In Simple Words

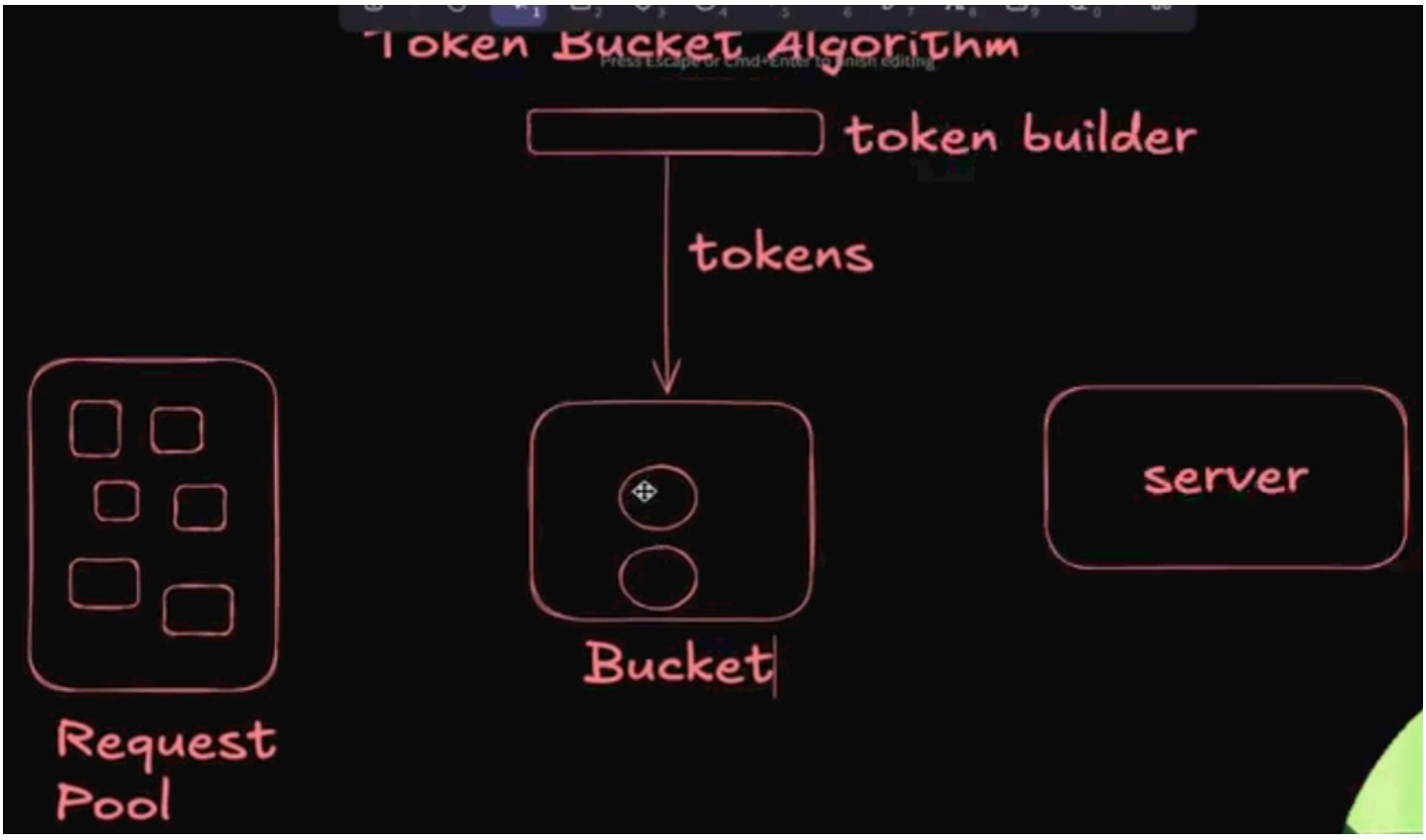
Each developer gets an API key.

That key has a fixed request limit.

If you cross the limit →
the API stops working for some time.



1. Token Bucket Algorithm



what it is:

Think of a bucket filled with tokens.

- **Tokens are added to the bucket at a fixed speed (for example: 2tokens per second).**
- **Every time a user makes a request, one token is removed from the bucket.**

How it works

- If there is a token in the bucket → request is allowed
- If the bucket is empty → request is blocked or delayed
- Extra tokens are saved in the bucket,
  but only up to the bucket's maximum size

Why we use it

- It controls the request rate
- It allows small bursts of traffic
- It keeps the system safe from overload

Simple Example

Bucket size = 3 tokens
Tokens added = 2 tokens per second

- User sends 1 requests → allowed (tokens available)
- If user sends too many and bucket becomes empty →
  new requests are blocked until tokens refill

Limitations of Token Bucket Algorithm

- It allows burst traffic, which can still stress the system if the burst is too big
- If the bucket size is large, many requests can come at once
- Not very good for strict, even request flow
- Needs extra memory to store tokens

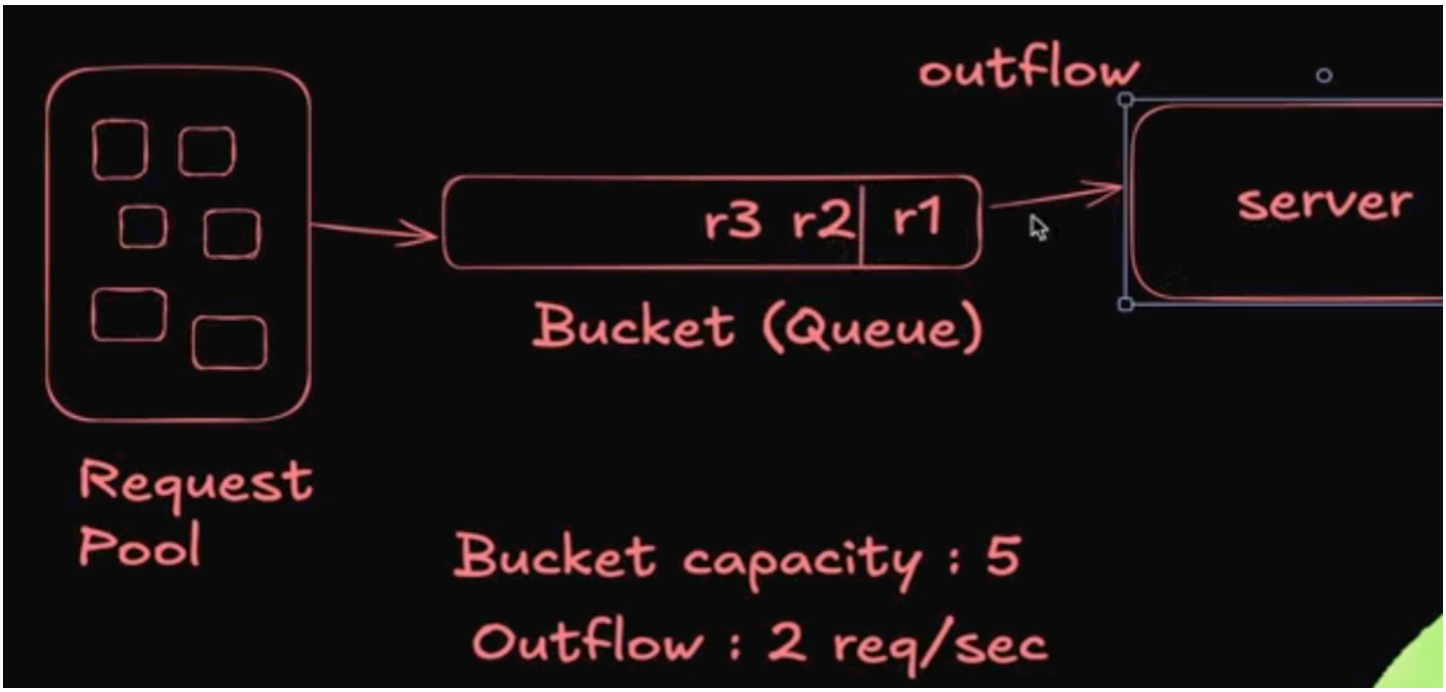
Pros

- Allows burst traffic (short sudden spikes are allowed)
- Controls average request rate
- Very flexible
- Good for real-world traffic
- Widely used in APIs and networks

Cons

- Burst traffic can still overload the system
- Not good for strict rate control
- Needs extra memory to store tokens
- More complex than simple counter method
- Harder to tune (bucket size + refill rate)

2.Leaky Bucket Algorithm



What it is

Think of a bucket with a small hole at the bottom.

- Requests go into the bucket
- Requests leave the bucket at a fixed, constant rate
- If the bucket is full → new requests are dropped

How it works

- Incoming requests are queued in the bucket
- Requests are processed one by one at a steady speed
- Bucket has a fixed capacity
- When capacity is full → extra requests are rejected

Simple Example

- Bucket size = 5requests
- Processing rate = 2 request per second

If:

- 2 requests come → all are stored and processed slowly
- 10 requests come → only 5 stored, 5 are dropped

Pros

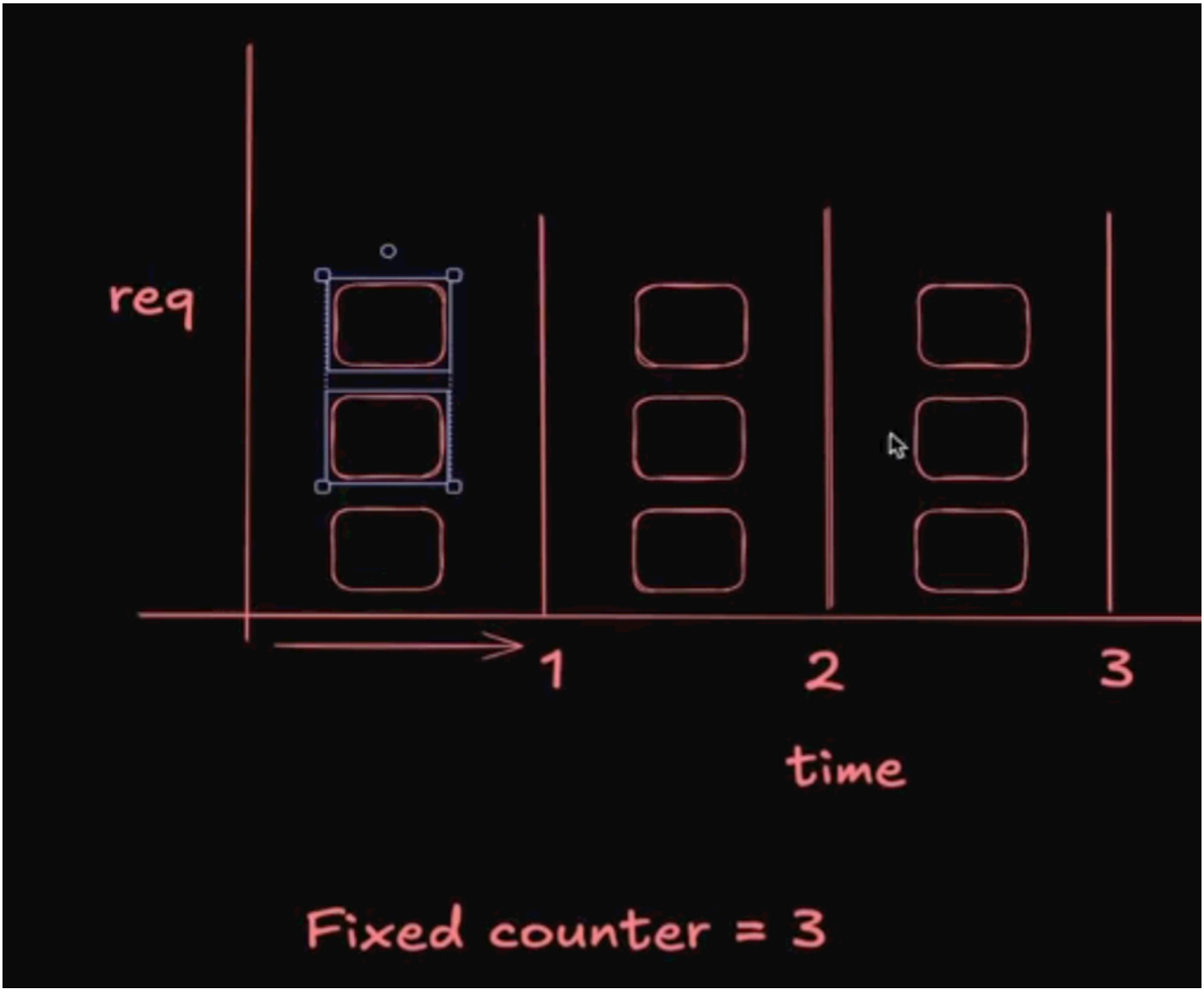
- Maintains a smooth and constant request flow
- Very good for strict rate limiting
- Prevents sudden traffic spikes
- Protects backend systems well
- server will not crash whatever my burst traffic.

Cons

- Does not allow bursts
- Some requests may be dropped even if system is free
- Can cause delay (latency)
- if DDOS/DOS attack hapens our server misses valuable request.

Leaky Bucket = Strict and smooth
Token Bucket = Flexible and burst-friendly

3.Fixed Window Counter Algorithm



What it is

Time is divided into fixed windows
(like 1 minute, 1 hour).

In each window, we count the number of requests.

How it works

- A counter starts at 0 for each time window
- Every request increases the counter
- If the counter exceeds the limit → request is blocked
- When the window ends → counter resets to 0

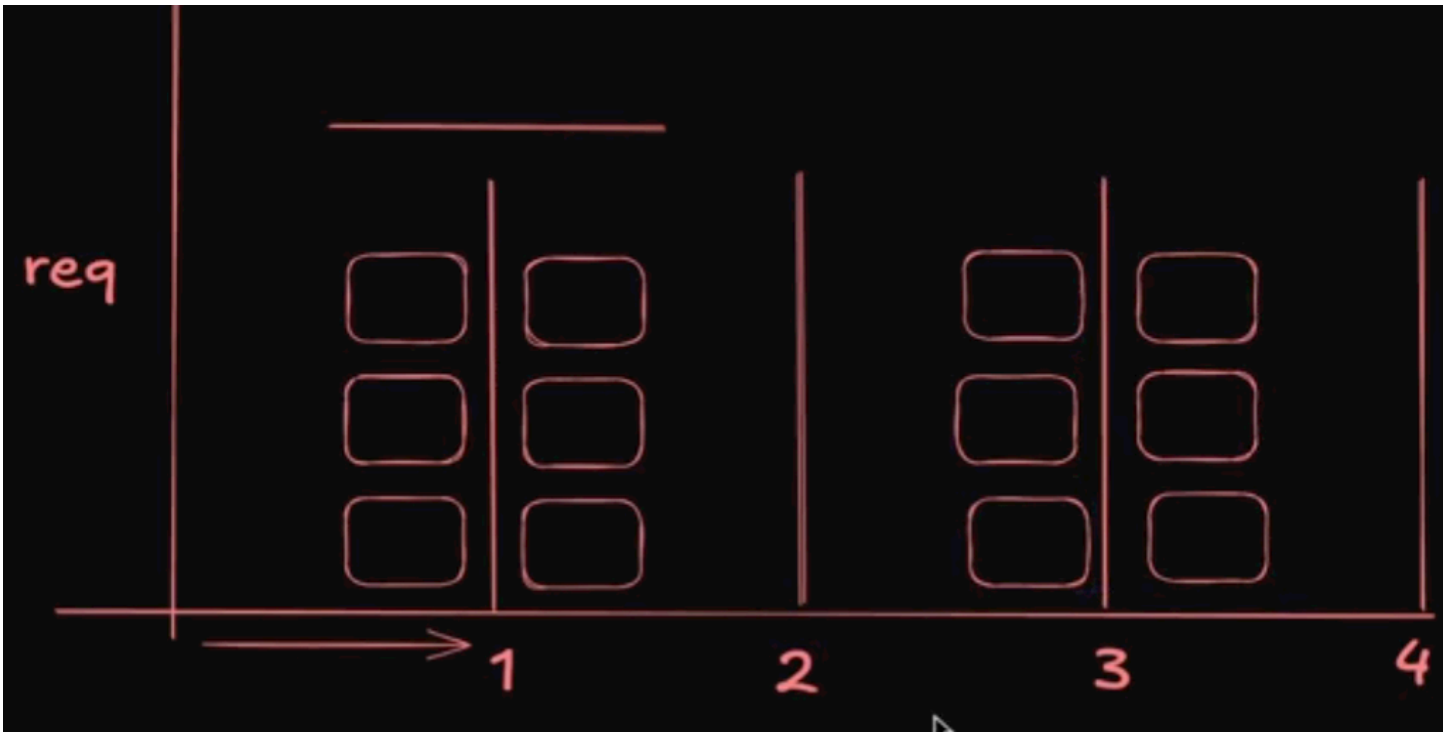
Simple Example

Limit = 100 requests per minute

- In one minute, user sends 80 requests → allowed
- User sends 120 requests → last 20 are blocked
- After 1 minute → counter resets, user can send again

Pros

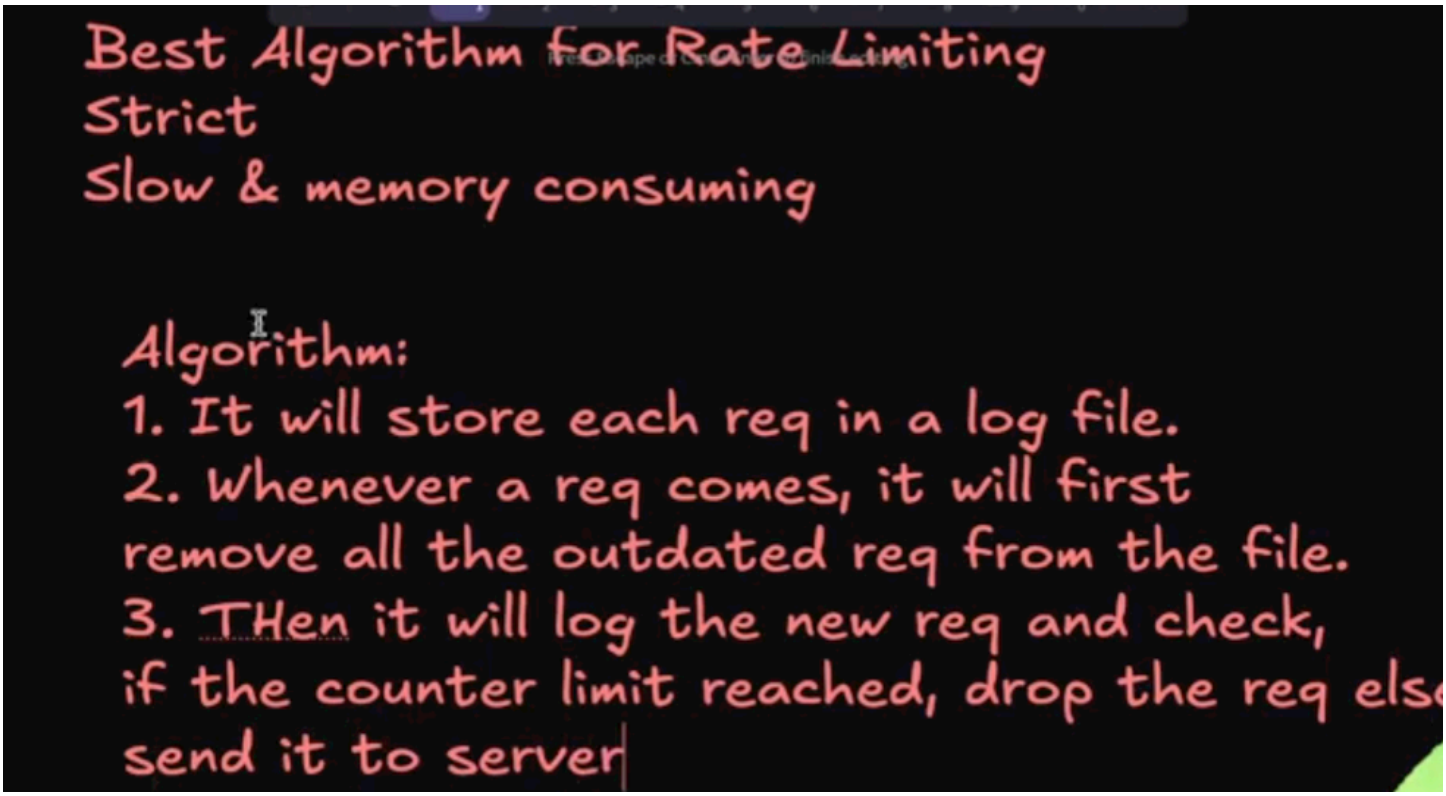
- Very simple to understand and implement
- Uses less memory
- Fast performance



Cons

- Allows traffic spikes at window boundaries
- Not accurate for real traffic flow
- Can overload system at window reset time

4.Sliding Window Log Algorithm



What it is

This algorithm tracks the exact time of every request inside a moving (sliding) time window.

How it works

- **Store the timestamp of each request**
- **For every new request:**
  1. **Remove timestamps that are outside the time window**
  2. **Count remaining timestamps**
  3. **If count ≤ limit → request allowed**
  4. **If count > limit → request blocked**

Simple Example

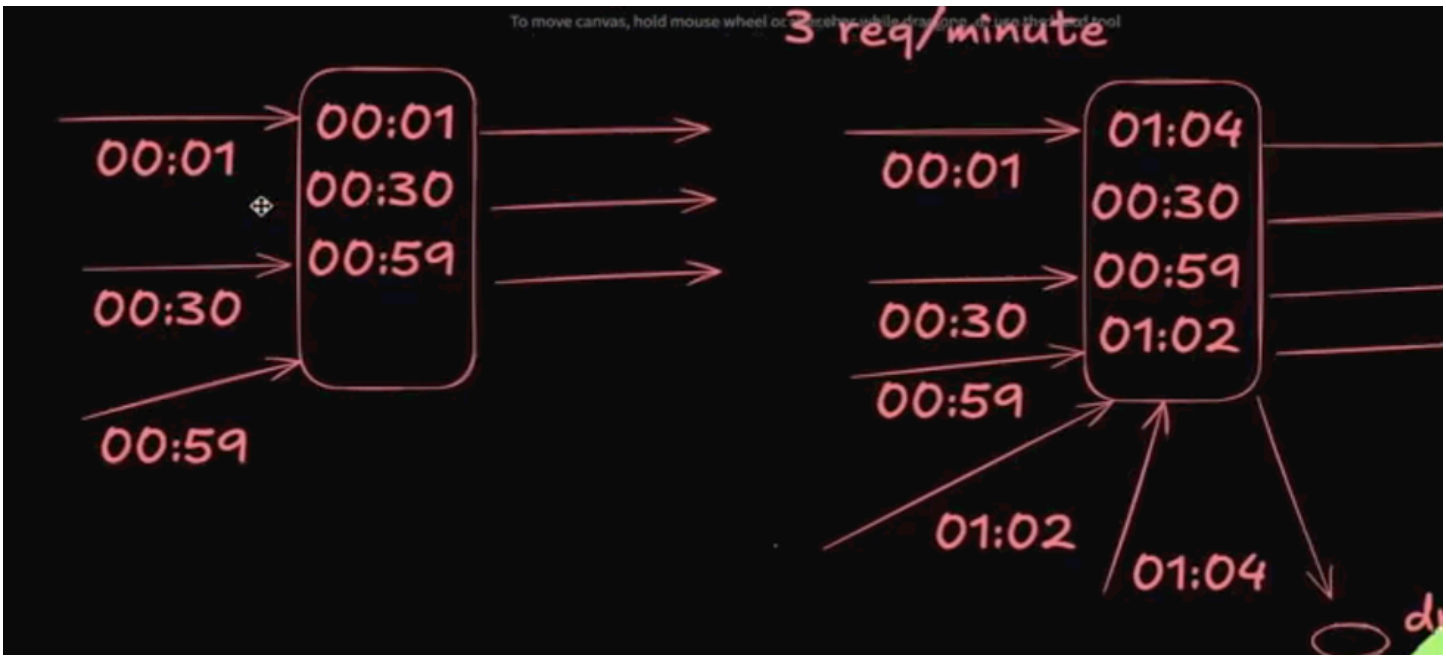
Limit = 5 requests per minute

Requests at times:

- **10:00:05**
- **10:00:10**
- **10:00:20**
- **10:00:30**
- **10:00:40 → allowed**
- **Epoch Timestamp**

Next request at:

- 10:00:45 → blocked
  (Already 5 requests in last 60 seconds)



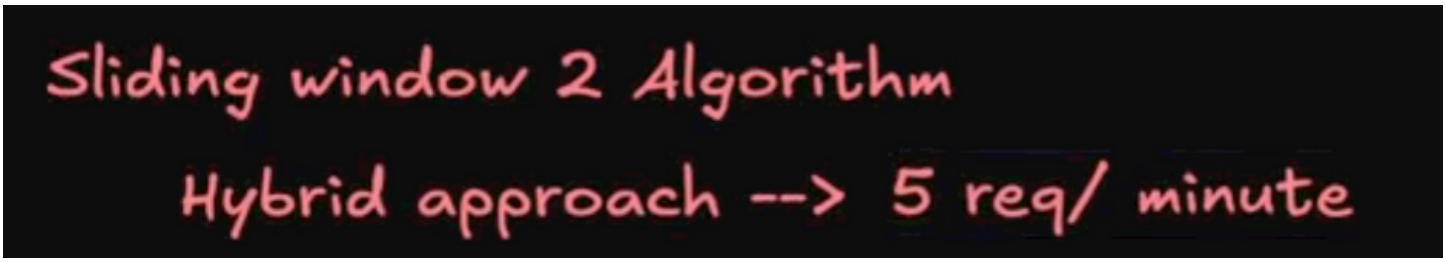
Pros

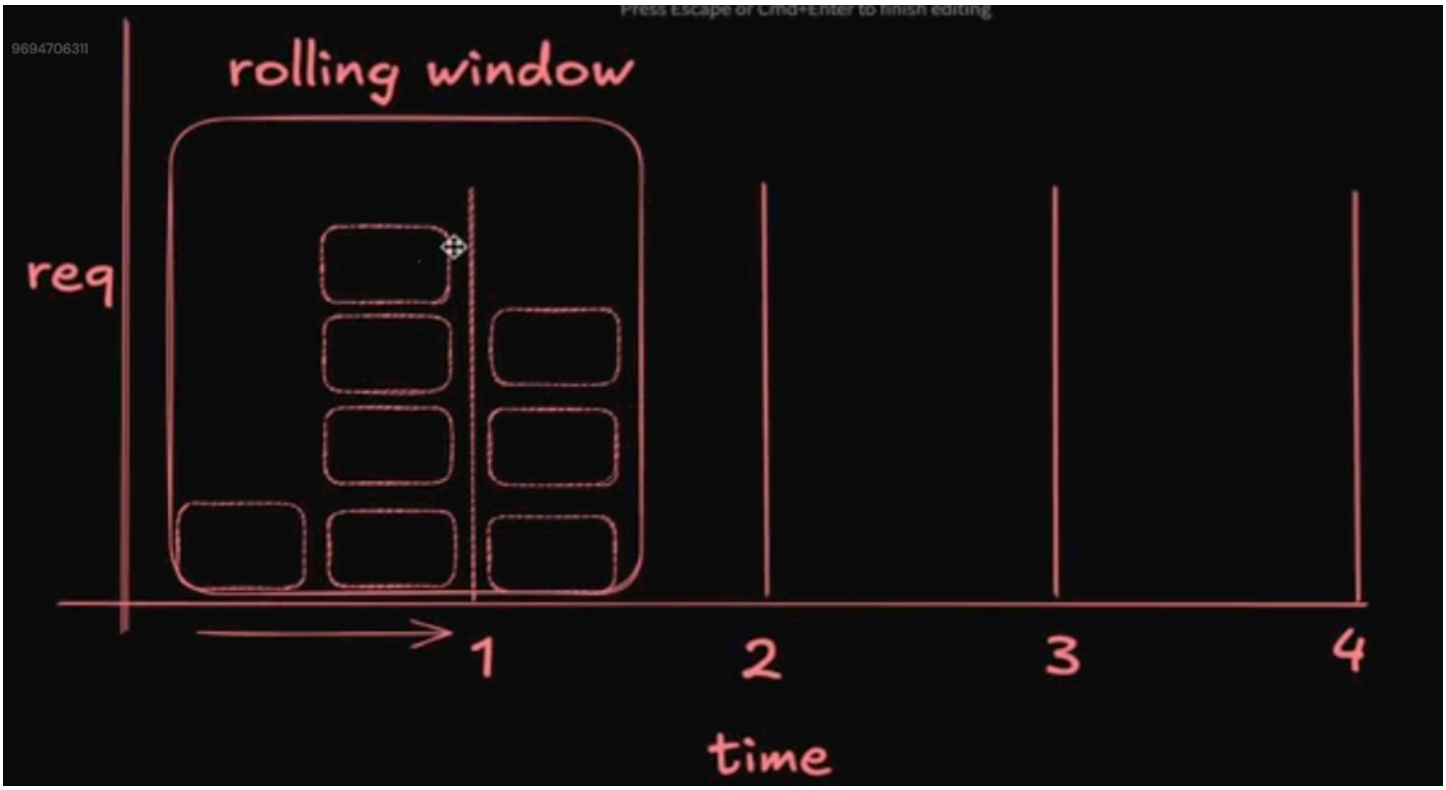
- Very accurate
- No sudden spikes
- Fair for all users
- Smooth traffic control

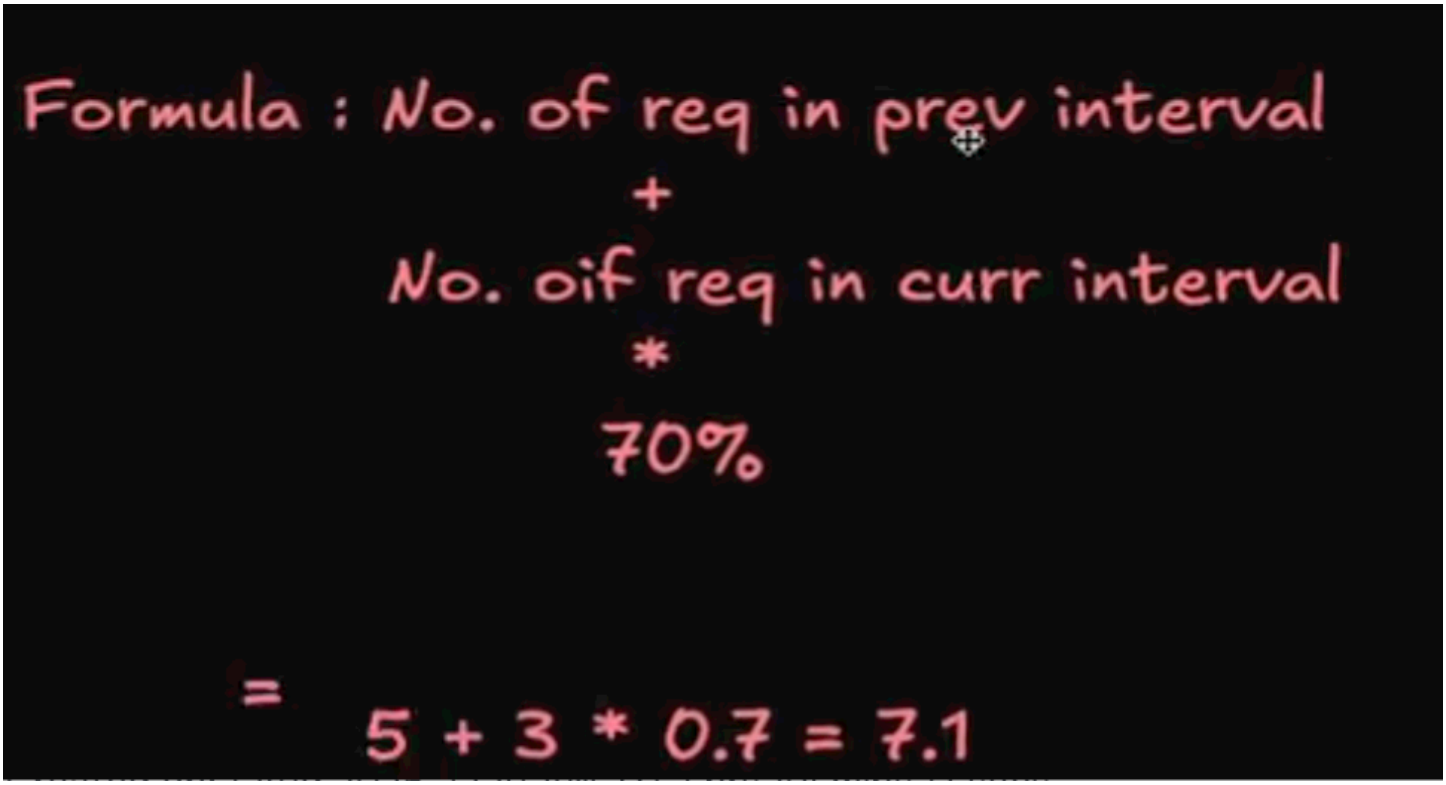
Cons

- Uses more memory (stores timestamps)
- Slower than counter-based methods
- Harder to implement at scale

5.Sliding Window Counter Algorithm (Hybrid Approch)





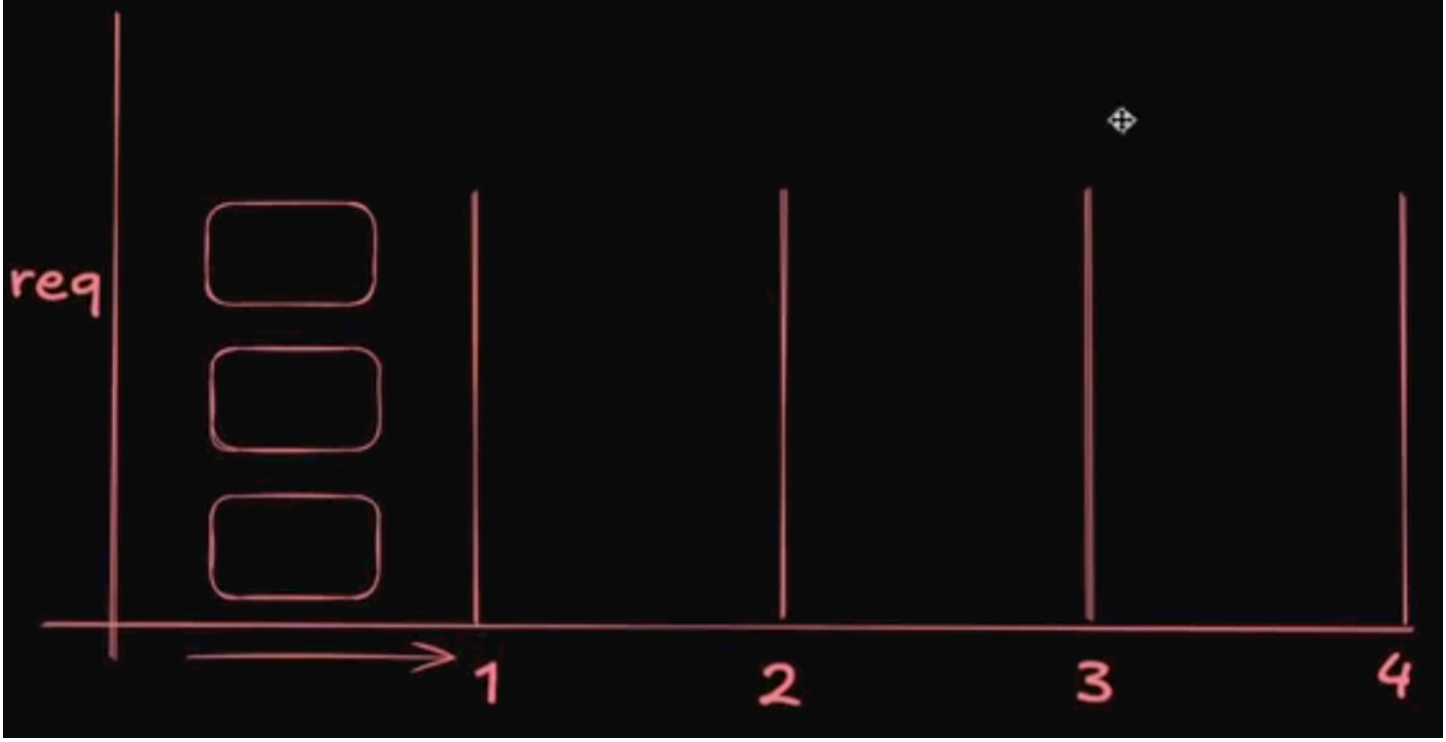


it is so complicated we rarely used this algo

Create your own Rate Limiter

Create our own Rate Limiter

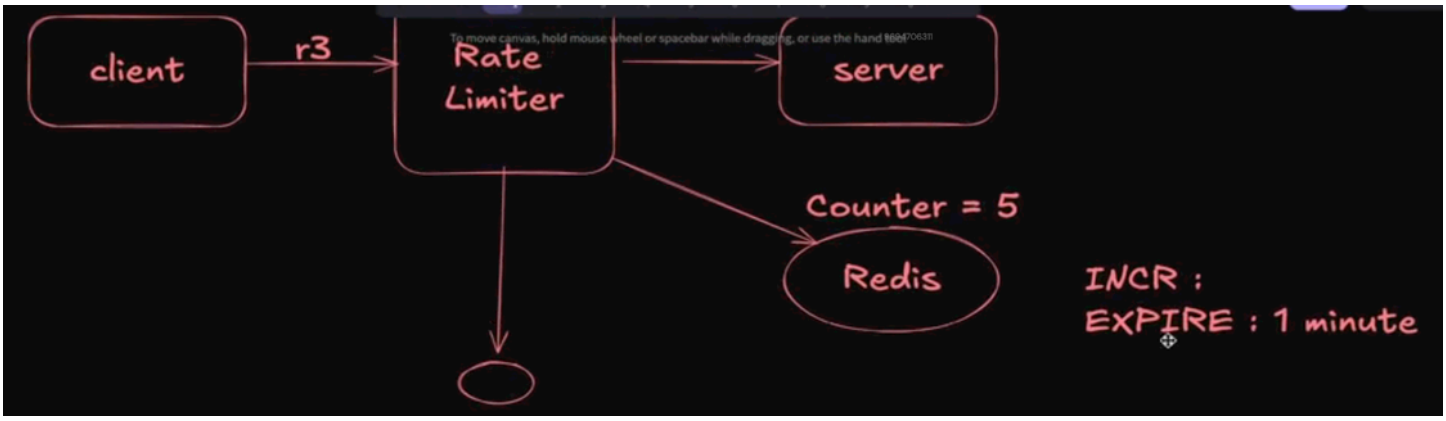
Which algo to implement ?
--> Fixed window counter

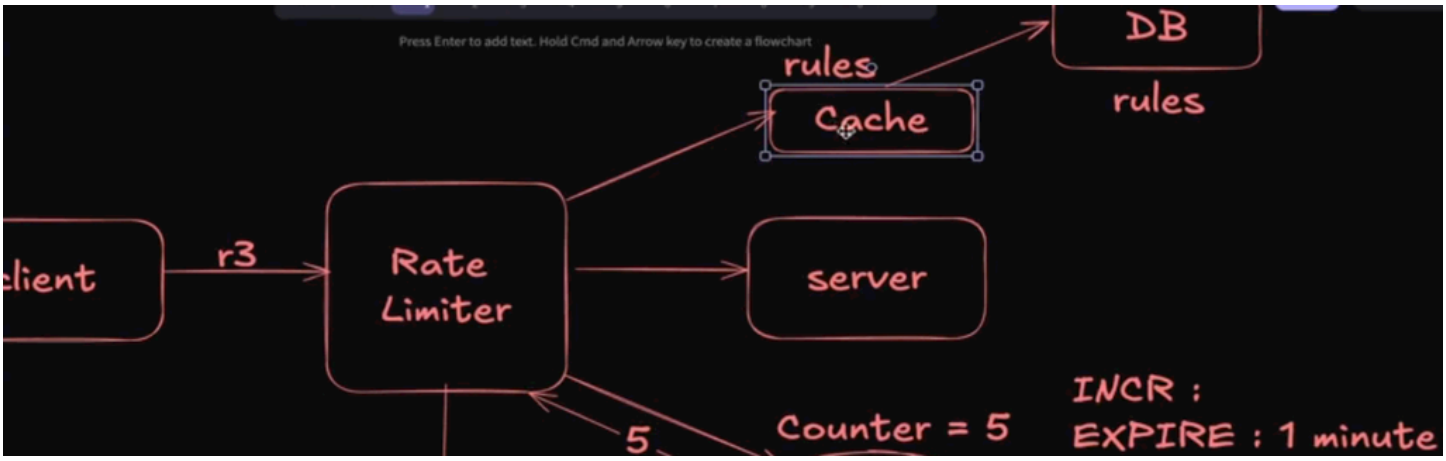


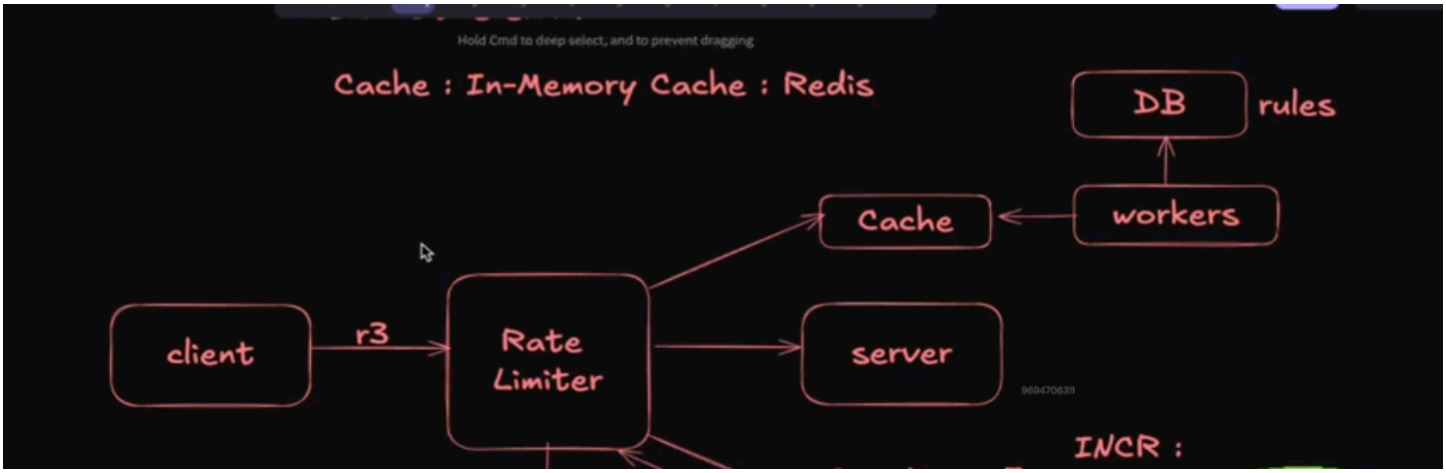
Db ? SQL / NoSQl

Db : It is slow.

Cache : In-Memory Cache : Redis







we want to make loosely coupled rate limiter