ADVANCED SYSTEM DESIGN

↳ Rate Limiting

→ Rate limiting :-

Rate Limiting is a technique to control the number of requests a user, client, or system can make to a server within a specific time period. It prevents abuse, overload, and server crashes.

Why it is used :-

Avoid DDoS attacks and bot spamming

Prevent resource exhaustion

Ensure fair usage among users

Maintain performance and stability of the system

How It Works?

You define a rule like:

> Max 100 requests per user per minute.

If a client exceeds the limit → **block / throttle / return 429 Too Many Requests** response.

Example:

yaml — Copy code

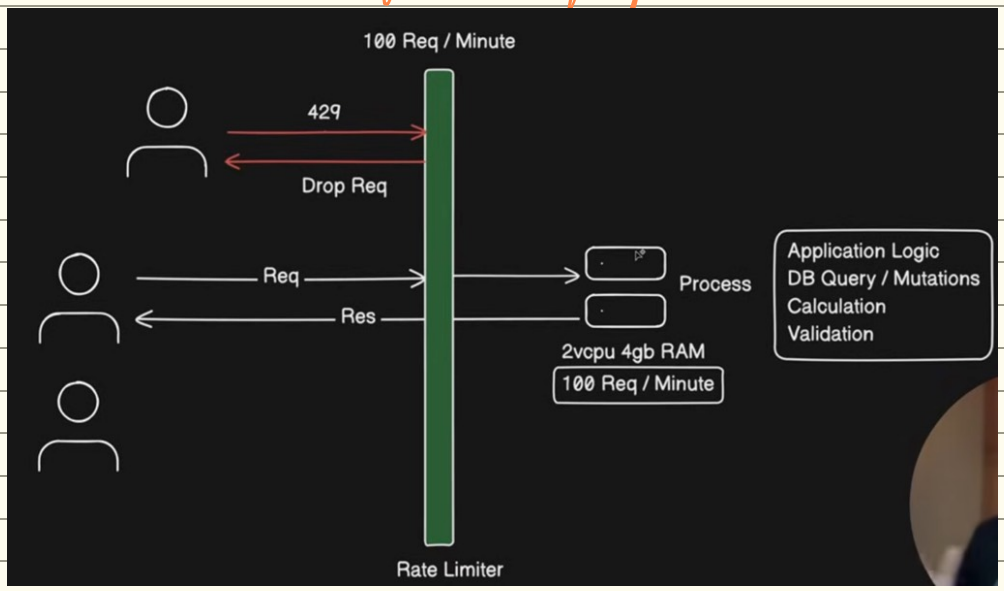
```
HTTP 429: Too Many Requests
Retry-After: 60 seconds
```

Popular Rate Limiting Strategies

Strategy	How it works	Pros	Cons
Fixed Window	Limit per fixed intervals (e.g., per minute)	Simple	Burst at window edges
Sliding Window Log	Tracks timestamps of each request	Very accurate	Memory heavy
Sliding Window Counter	Smooth decreasing counter	More fair	Complex implementation
Token Bucket	Tokens refill at a rate; consume per request	Allows bursts, widely used (APIs)	Needs token tracking
Leaky Bucket	Queue leaks at constant rate	Smooth traffic flow	Can drop sudden requests

Error Code :- 429, "TOO MANY REQUESTS", HTTP status code indicating that the client has sent too many requests to the server within a given time frame.

High level design of a Rate limiter.



→ To implement Rate limiting, we have several algorithms. :-

- Token Bucket
- Leaking Bucket.
- Fixed Window
- Sliding Window log.
- Sliding window counter.

→ Token Bucket Algorithm :-

The Token Bucket Algorithm is a popular rate limiting technique used in computer networks and applications to control the amount of data or number of requests allowed over time.

→ It ensures that a system:

- ✓ does not get overloaded
- ✓ stays responsive
- ✓ allows bursts of traffic briefly while still enforcing an average limit

→ 🌑 Concept

Imagine a bucket that has tokens added to it at a fixed rate (e.g., 10 tokens per second).

Each request/data packet needs 1 token to be processed.

If the bucket has tokens → request is allowed (remove a token).

If the bucket is empty → request is blocked or queued.

⚙ Parameters

Parameter	Meaning
Token generation rate (r)	Number of tokens added per second
Bucket capacity (b)	Maximum number of tokens bucket can hold

📌 Example

Suppose:

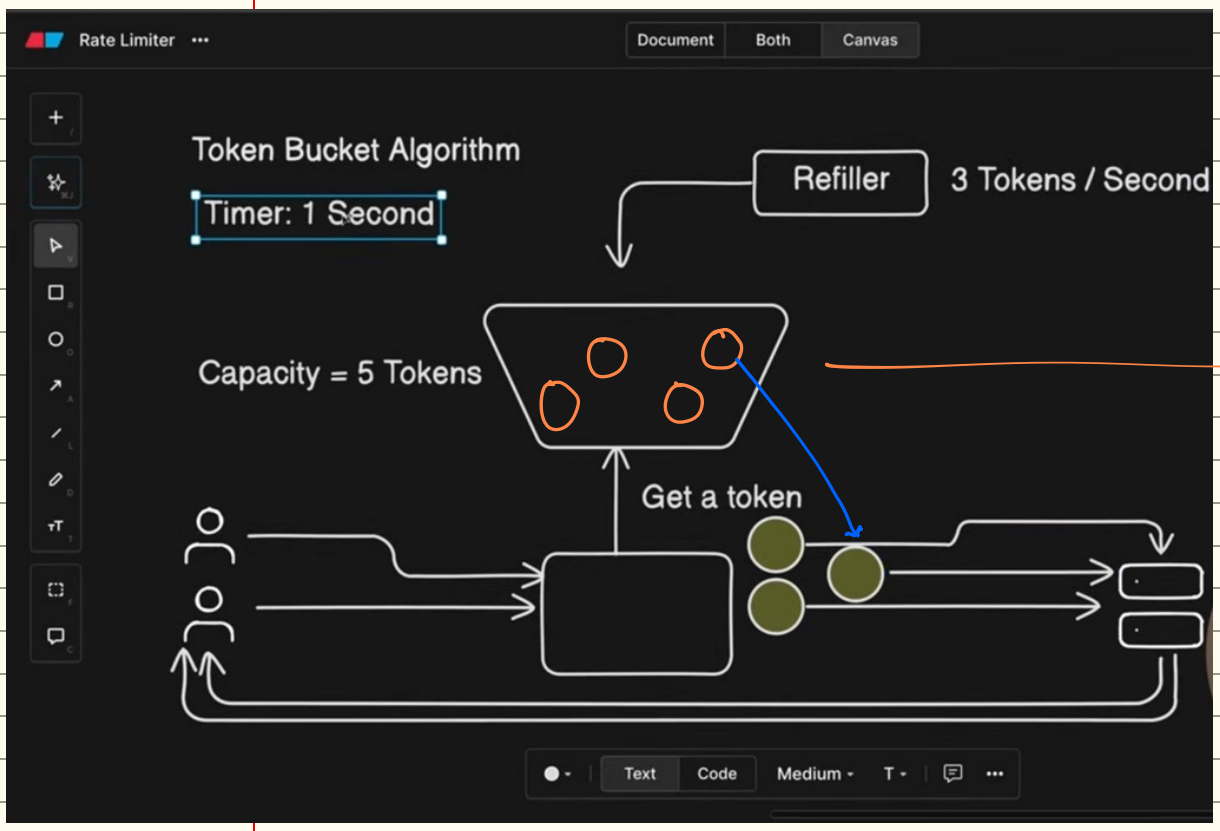
- Bucket size = **20 tokens**
- Token rate = **5 per second**

You can send:

- **Up to 20 requests instantly** (burst allowed)
- Then **only 5 requests per second** on average

If more requests arrive:

- With tokens → allowed
- Without tokens → dropped/queued

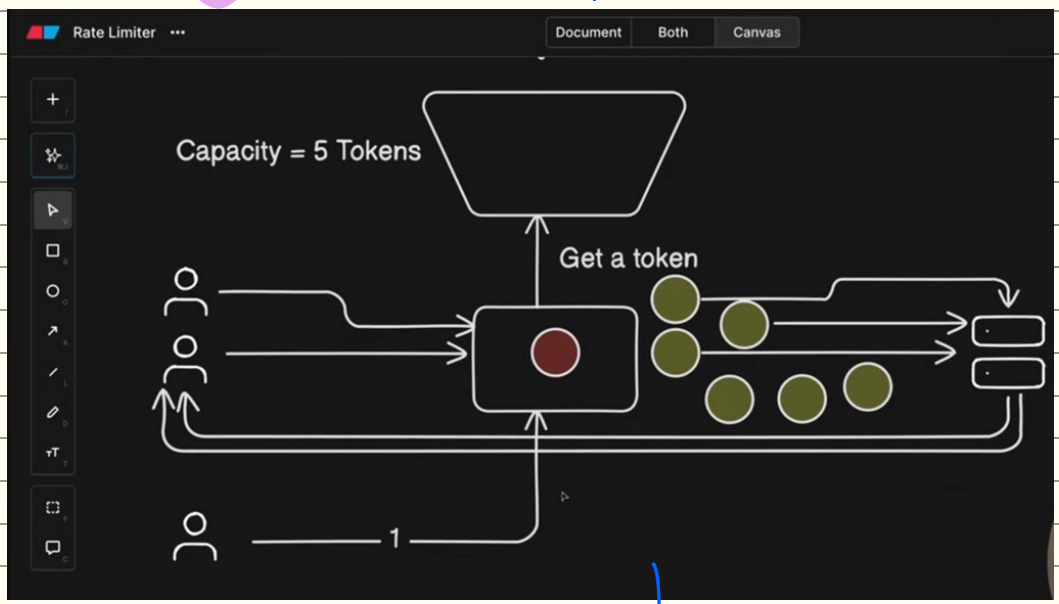


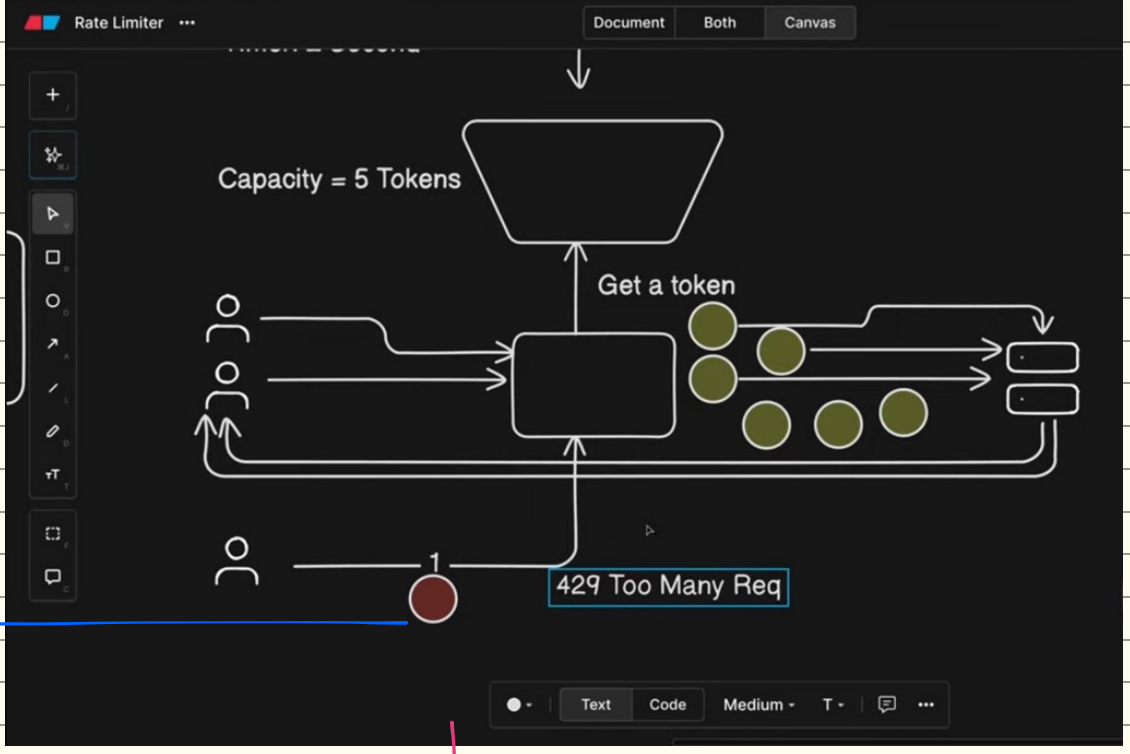
→ Bucket containing available tokens.

∴ Tokens are removed and used to process user request.

∴ The refiller refills new tokens in the bucket after the tokens get exhausted.

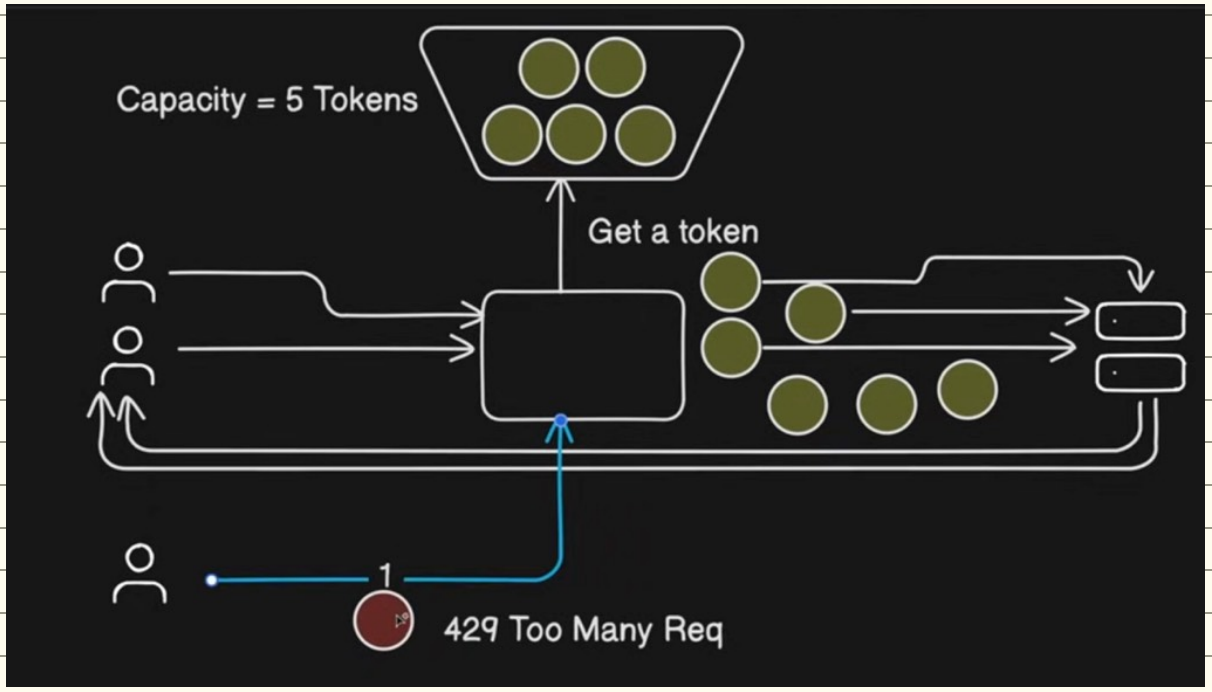
When the tokens are busy and not available





The request is rejected

Now, in the next second, the refiller again refills the bucket inside the token.

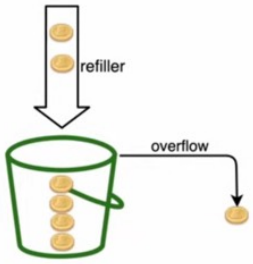


Token bucket algorithm

The token bucket algorithm is widely used for rate limiting. It is simple, well understood and commonly used by internet companies. Both Amazon [5] and Stripe [6] use this algorithm to throttle their API requests.

The token bucket algorithm work as follows:

- A token bucket is a container that has pre-defined capacity. Tokens are put in the bucket at preset rates periodically. Once the bucket is full, no more tokens are added. As shown in Figure 4, the token bucket capacity is 4. The refiller puts 2 tokens into the bucket every second. Once the bucket is full, extra tokens will overflow.



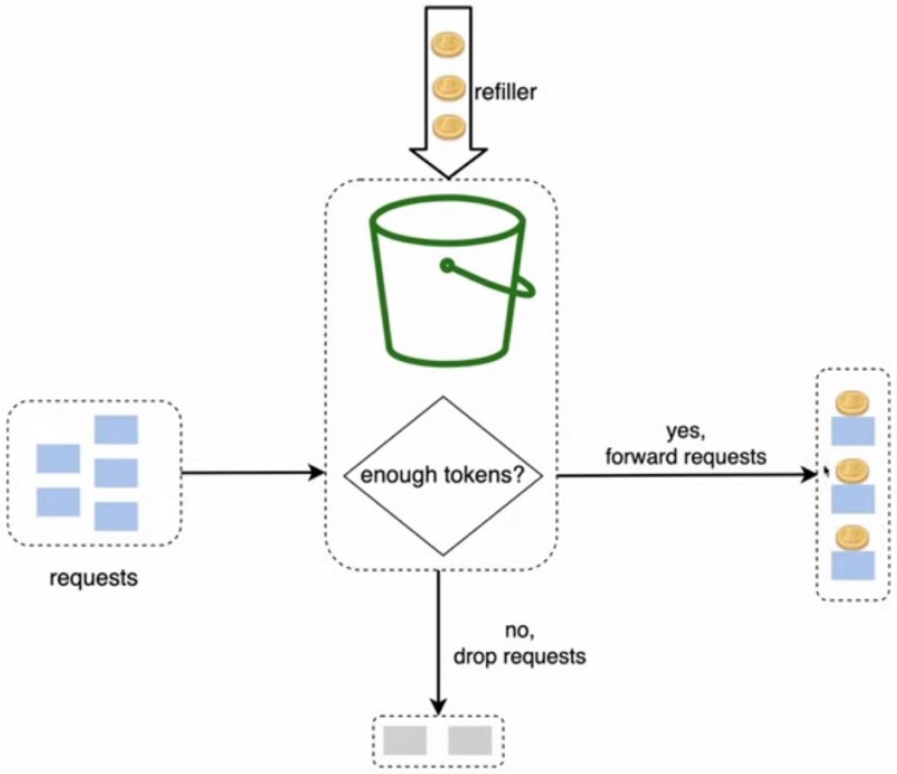


Figure 5

Pros:

- The algorithm is easy to implement.
- Memory efficient.
- Token bucket allows a burst of traffic for short periods. A request can go through as long as there are tokens left.

Cons:

- Two parameters in the algorithm are bucket size and token refill rate. However, it might be challenging to tune t

→ Leaky Buck Algorithm (Works on a queue system):-

The Leaky Bucket Algorithm is another rate limiting and traffic shaping technique used in networks and application services to enforce a fixed, constant output rate.

Core Idea

Think of a bucket with a **small hole at the bottom**:

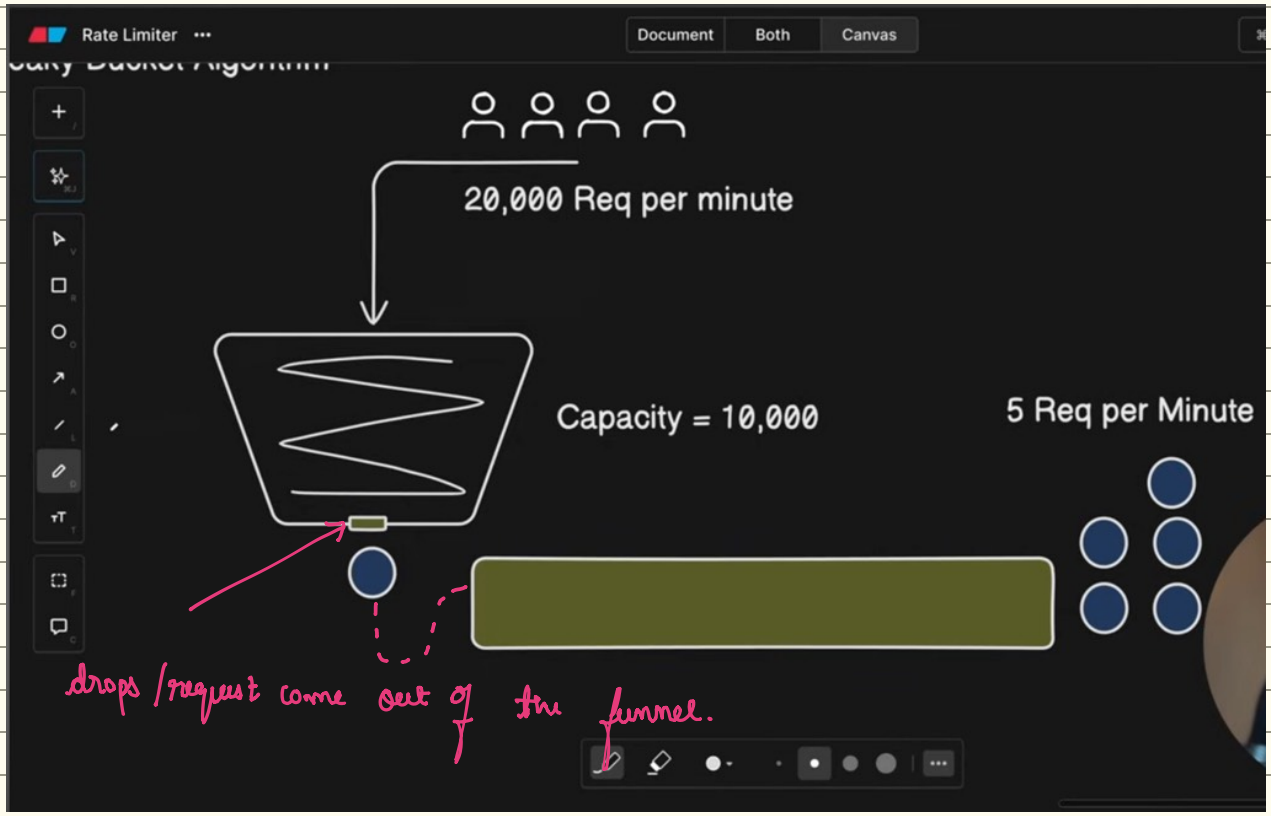
- Water (requests/data packets) comes into the bucket at variable speed.
- Water leaks out **at a steady, fixed rate.**
- If too much water comes in and the bucket overflows → **extra requests are dropped.**

How It Works

Input Traffic (Burst allowed?)	Output Traffic
Variable traffic → bursty	**Uniform and constant**

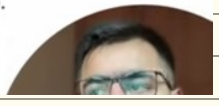
Algorithm Behavior

- Packets/requests arrive: **queued in the bucket**
- Output: **one packet per fixed interval**
- If queue is full → **incoming packets are discarded**



The leaking bucket algorithm is similar to the token bucket except that requests are processed at a fixed rate. It is usually implemented with a first-in-first-out (FIFO) queue. The algorithm works as follows:

- When a request arrives, the system checks if the queue is full. If it is not full, the request is added to the queue.
- Otherwise, the request is dropped.
- Requests are pulled from the queue and processed at regular intervals.



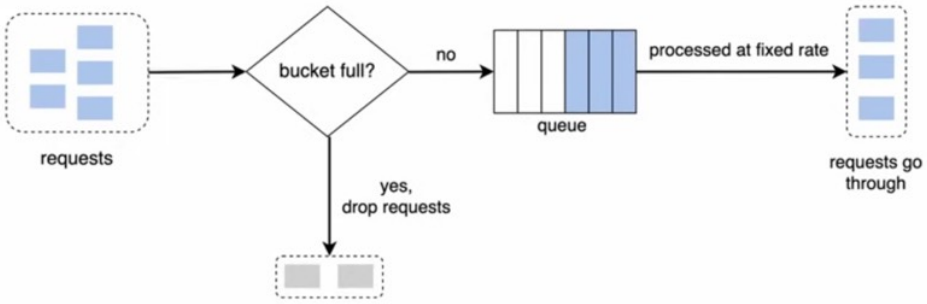


Figure 7

Leaking bucket algorithm takes the following two parameters:

- Bucket size: it is equal to the queue size. The queue holds the requests to be processed at a fixed rate.
- Outflow rate: it defines how many requests can be processed at a fixed rate, usually in seconds.

Shopify, an ecommerce company, uses leaky buckets for rate-limiting [7].

Pros:

- Memory efficient given the limited queue size.
- Requests are processed at a fixed rate therefore it is suitable for use cases that a stable outflow rate is needed.

Cons:

- A burst of traffic fills up the queue with old requests, and if they are not processed in time, recent requests will be rate limited.
- There are two parameters in the algorithm. It might not be easy to tune them properly.

→ Fixed Window Counter Algorithm :-

The Fixed Window Counter Algorithm is a simple rate limiting method used in APIs and distributed systems to restrict the number of requests a user can make within a fixed time window.

🧠 Concept

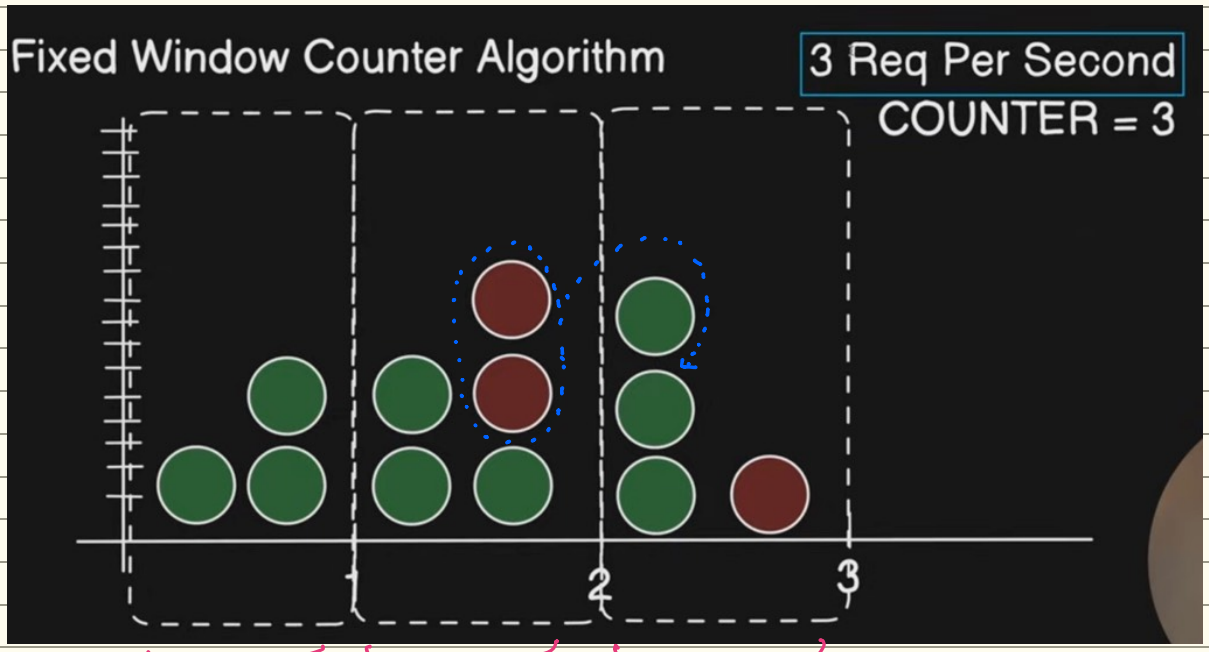
Time is divided into **equal fixed intervals** (windows), such as:

- 1 second
- 1 minute
- 1 hour

A **counter** tracks how many requests a client makes in the current window.

Rule:

If the request count **> limit**, block or throttle requests
Else accept and **increment** the counter.



fixed window size with a determined threshold.

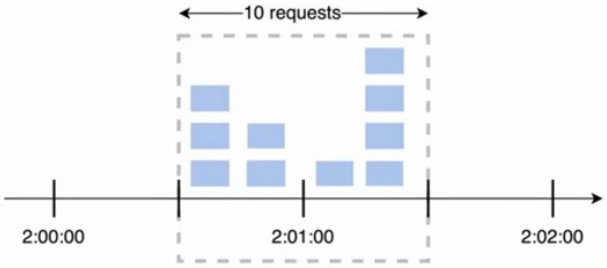
Fixed window counter algorithm works as follows:

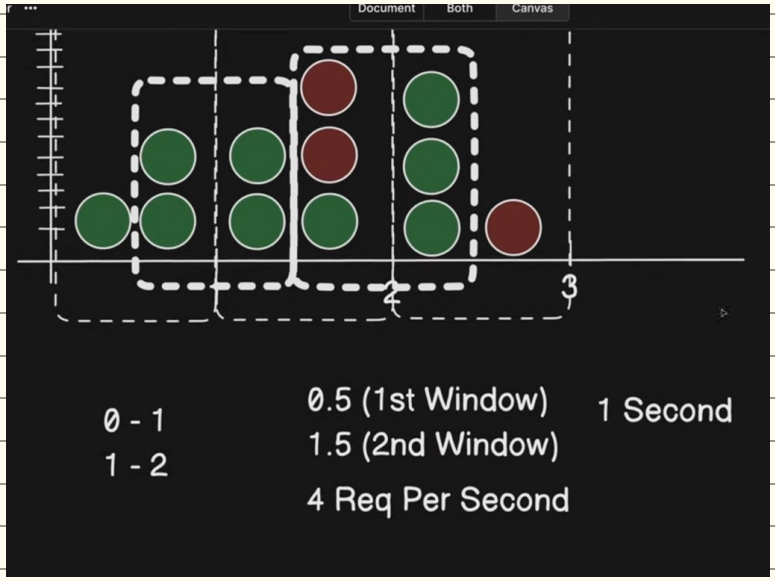
- The algorithm divides the timeline into fix-sized time windows and assign a counter for each window.
- Each request increments the counter by one.
- Once the counter reaches the pre-defined threshold, new requests are dropped until a new time window starts.

Let us use a concrete example to see how it works. In Figure 8, the time unit is 1 second and the system allows a maximum of 3 requests per second. In each second window, if more than 3 requests are received, extra requests are dropped as shown in Figure 8.

CONS OF FIXED WINDOW COUNTER.

A major problem with this algorithm is that a burst of traffic at the edges of time windows could cause more requests than allowed quota to go through. Consider the following case:





Pros:

- Memory efficient.
- Easy to understand.
- Resetting available quota at the end of a unit time window fits certain use cases.

Cons:

- Spike in traffic at the edges of a window could cause more requests than the allowed quota to go through.

→ Sliding window log Algorithm :-

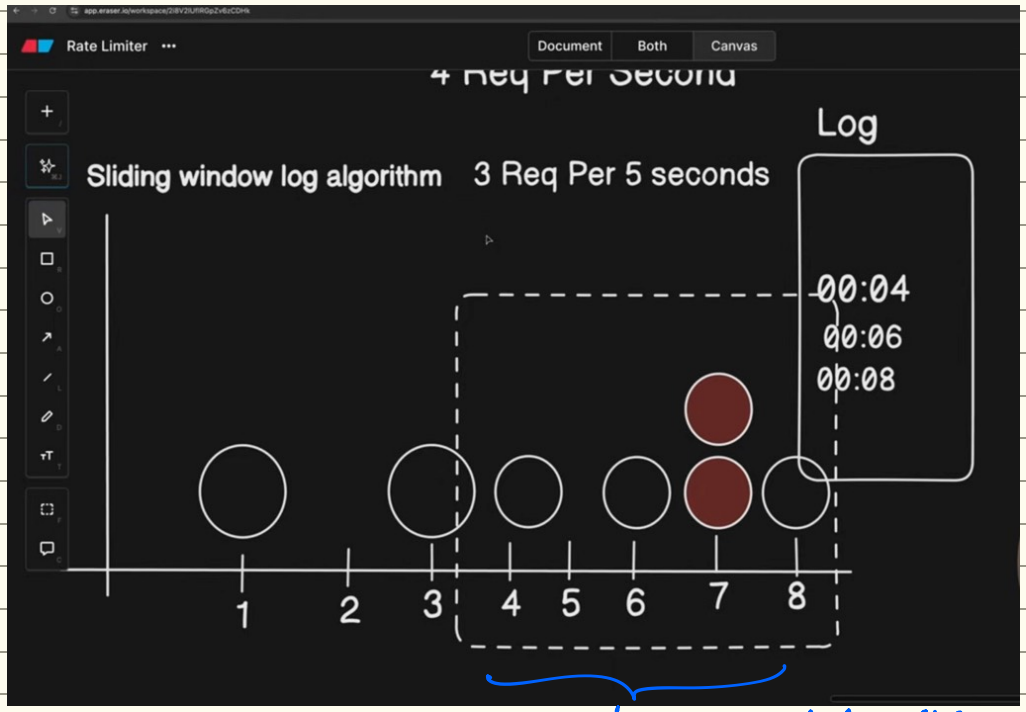
The Sliding Window Log Algorithm is an improved rate limiting technique that solves the burst problem of the fixed window method. It provides smooth and accurate request limiting over time.

🧠 Core Idea

Instead of counting requests in a fixed block of time, this algorithm:

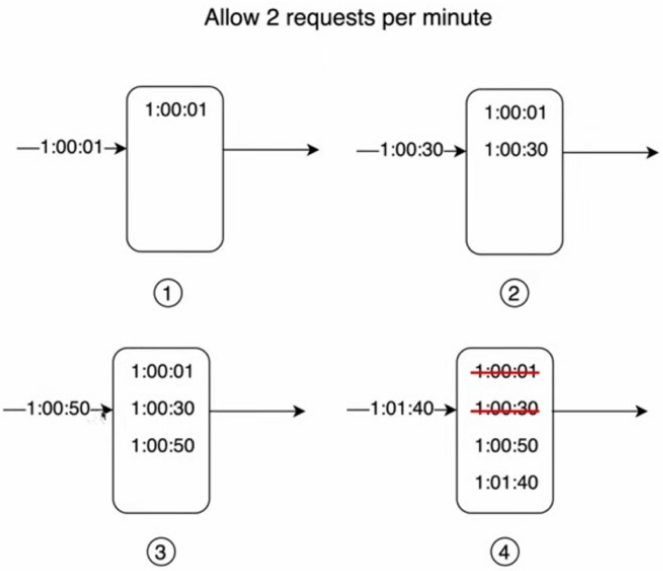
- ✓ Keeps a **timestamp log** of every request
- ✓ Always checks the **last X time duration** (sliding window)
- ✓ Deletes timestamps older than the window

👉 So the limit applies to a **continuous** time window, not a fixed block.



Sliding window slides and expires previous timers.

We explain the algorithm with an example as revealed in Figure 10.



As discussed previously, the fixed window counter algorithm has a major issue: it allows more requests to go through at the edges of a window. The sliding window log algorithm fixes the issue. It works as follows:

- The algorithm keeps track of request timestamps. Timestamp data is usually kept in cache, such as sorted sets of Redis [8].
- When a new request comes in, remove all the outdated timestamps. Outdated timestamps are defined as those older than the start of the current time window.
- Add timestamp of the new request to the log.
- If the log size is the same or lower than the allowed count, a request is accepted. Otherwise, it is rejected.

We explain the algorithm with an example as revealed in Figure 10.

→ Sliding Window Counter Algorithm :-

↳ Sliding Window + Counter Algorithm.

The **Sliding Window Counter Algorithm** is a **hybrid rate limiting** technique — it combines concepts from:

- **Fixed Window Counter** (simple + low memory)
- **Sliding Window Log** (smooth + accurate)

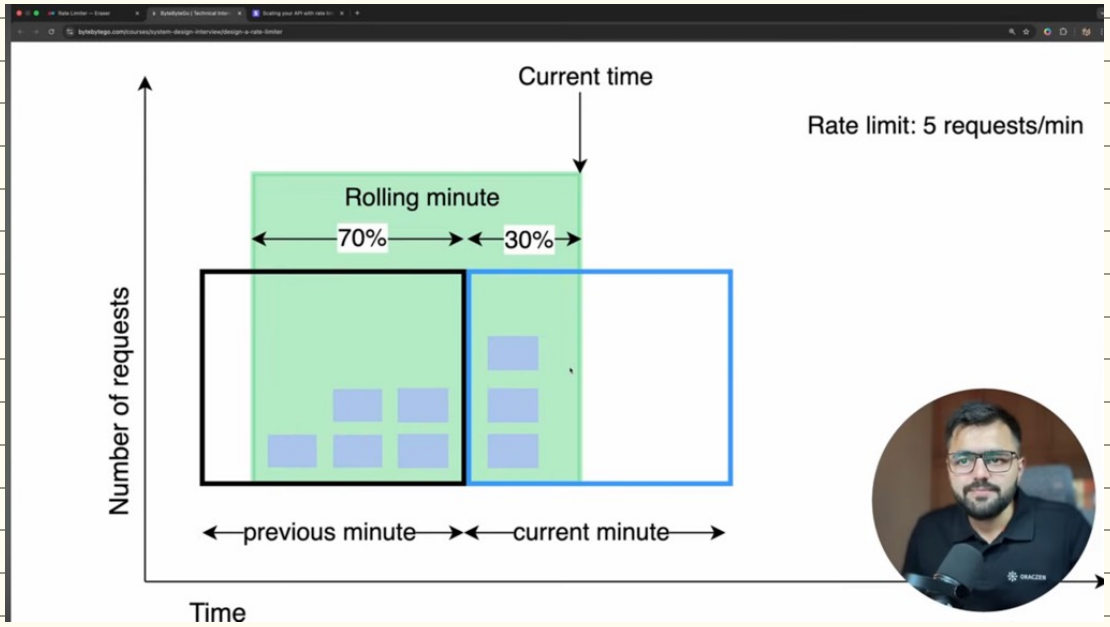
Its goal is to **reduce bursts** while keeping the algorithm efficient.

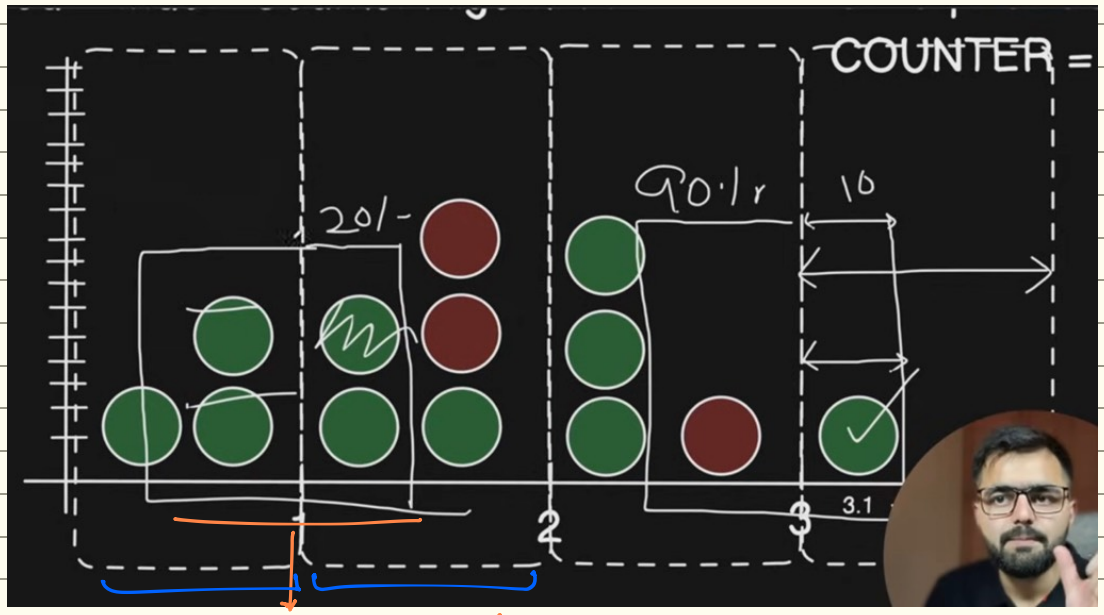
🧠 Idea

Instead of using one fixed time window, this algorithm uses:

- The **current window**
- A **portion** of the **previous window**

The rate is **proportionally calculated** based on how much of the previous window overlaps with the current time.





Sliding window inside
fixed window