#Rate Limiting

Rate limiting is a popular distributed system pattern. It is an integral part of all modern large-scale applications. It controls the rate at which users or services can access a resource, like an API, a service, or a network. It plays a critical role in protecting system

Rate Limiting Fundamentals

Rate limiting controls the rate at which users or services can access a resource. When the rate of requests exceeds the threshold defined by the rate limiter, the requests are throttled or blocked. Here are some examples:

1. A user can send a message no more than 2 per second
2. One can create a maximum of 10 accounts per day from the same IP address
3. One can claim rewards no more than 5 times per week from the same device

Benefits of Rate Limiting

Rate limiting is an integral part of modern large-scale applications. Let's take a look at some of the benefits of a rate limiter. While we focus on API rate limiters here, the benefits are generally applicable to rate limiters deployed for other use cases.

Prevent Resource Starvation

Rate limiting helps prevent resource starvation caused by Denial of Service (DoS) attacks. Almost all APIs published by large tech companies enforce some form of rate limiting. For example, Twitter limits the number of tweets to 300 per 3 hours. Google docs APIs limit read requests to 300 per user per 60 seconds. A rate limiter prevents DoS attacks, either intentional or unintentional, by rejecting the excess calls.

Reduce cost

Rate limiting can help limit cost overruns by preventing the overuse of a resource. If a resource is overloaded by a high volume of requests, it may require additional resources or capacity to handle the load, which can incur additional costs.

Rate limiting is also critical for services that make outbound requests, especially those that use paid third party APIs. Many third-party services charge on a per-call basis for their external APIs. Some examples are services for checking credit, making a payment, retrieving health records, etc. Limiting the number of calls is essential to control costs.

Prevent servers from being overloaded.

While rate limiting is vital in preventing DoS attacks, it also plays a pivotal role in general load balancing and service quality maintenance. High volumes of requests, not only from malicious sources but also from heavy usage, can overburden servers.

To reduce server load, a rate limiter is used to reject excess requests early in the request lifecycle made by malicious bots or users with heavy usage.

Applications of Rate Limiting

Rate limiting can be used in various situations to manage resources efficiently. When done right, it prevents abuse and ensures fair usage.

Rate limiting is commonly applied at the user level. Consider a popular social media platform where users frequently post content and comments. To prevent spam or malicious bot activity, the platform might enforce user-level rate limiting. It restricts the number of posts or comments that a user can make in a given hour.

Rate limiting can also be applied at the application level. One example is an online ticketing platform. On the day of a major concert sale, the platform can expect a significant surge in traffic. Application-level rate limit can be very useful in this case. It limits the total number of ticket purchases per minute. This practice protects the system from being overwhelmed and ensures a fair chance for everyone to try to secure a ticket.

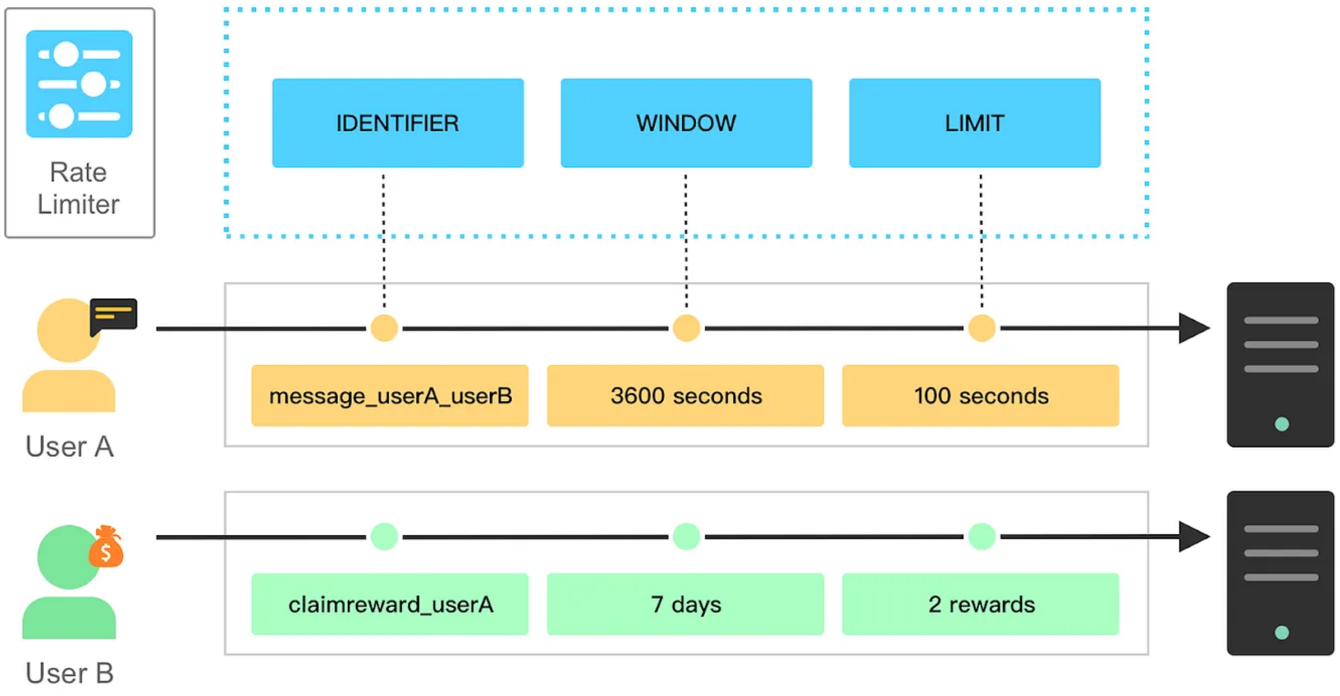
API-level rate limiting is also common. Consider a cloud storage service that provides an API for uploading and downloading files. To ensure fair use and protect the system from misuse, the service might enforce limits on the number of API calls each user can make per minute.

Rate limiting can also be applied based on user account levels. A SaaS platform offering multiple tiers of service can have different usage limits for each tier. Free tier users may have a lower rate limit compared to premium tier users. This effectively manages resource usage while encouraging users to upgrade to higher limits.

These are just a few examples. They demonstrate how rate limiting strategies can be deployed across different scenarios and

Core Concepts of Rate Limiting

Most rate limiting implementations share three core concepts. They are the limit, the window, and the identifier.
The limit defines the ceiling for allowable requests or actions within a designated time span. For example, we might allow a user to send no more than 100 messages every hour.
The window is the time period where the limit comes into play. It could be any length of time, whether it be an hour, a day, or a week. Longer durations do have their own implementation challenges, like storage durability, that we'll discuss later.
The identifier is a unique attribute that differentiates between individual callers. A user ID or IP address is a common example.

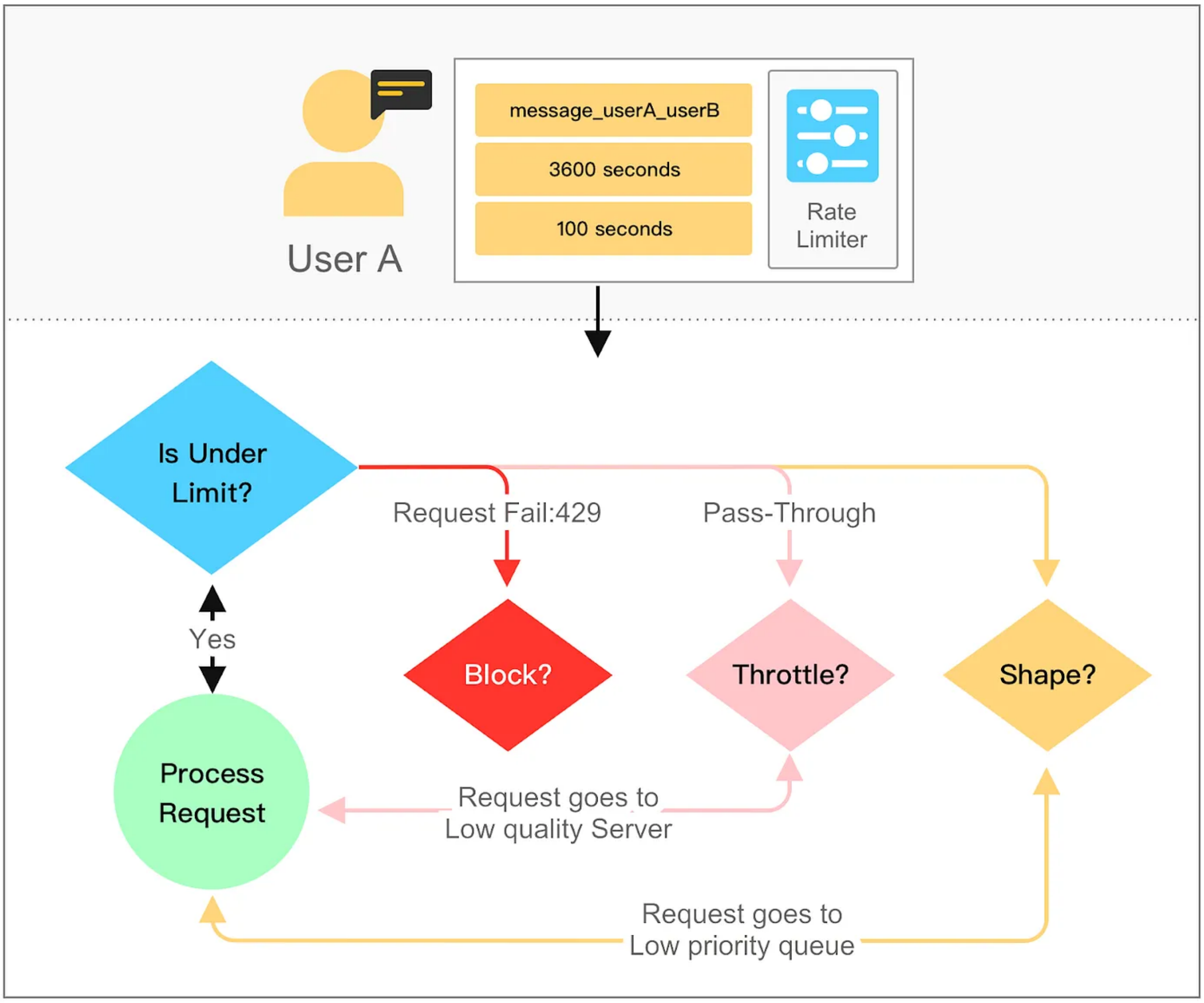


Another core concept to understand is the different types of rate limiting responses. They generally fall into three categories: **blocking**, **throttling**, and **shaping**.

Blocking takes place when requests exceeding the limit are denied access to the resource. It is commonly expressed as an error message such as HTTP status code 429 (Too Many Requests).

Throttling, by comparison, involves slowinging down or delaying the requests that go beyond the limit. An example would be a video streaming service reducing the quality of the stream for users who have gone over their data cap.

Shaping, on the other hand, allows requests that surpass the limit. But those requests are assigned lower priority. This ensures that users who abide by the limits receive quality service. For example, in a content delivery network, requests from users who have crossed their limits may be processed last, while those from normal users are prioritized.



These basic core concepts lay the foundation for diving into the various common rate limiting algorithms.

Common Rate Limiting Algorithms

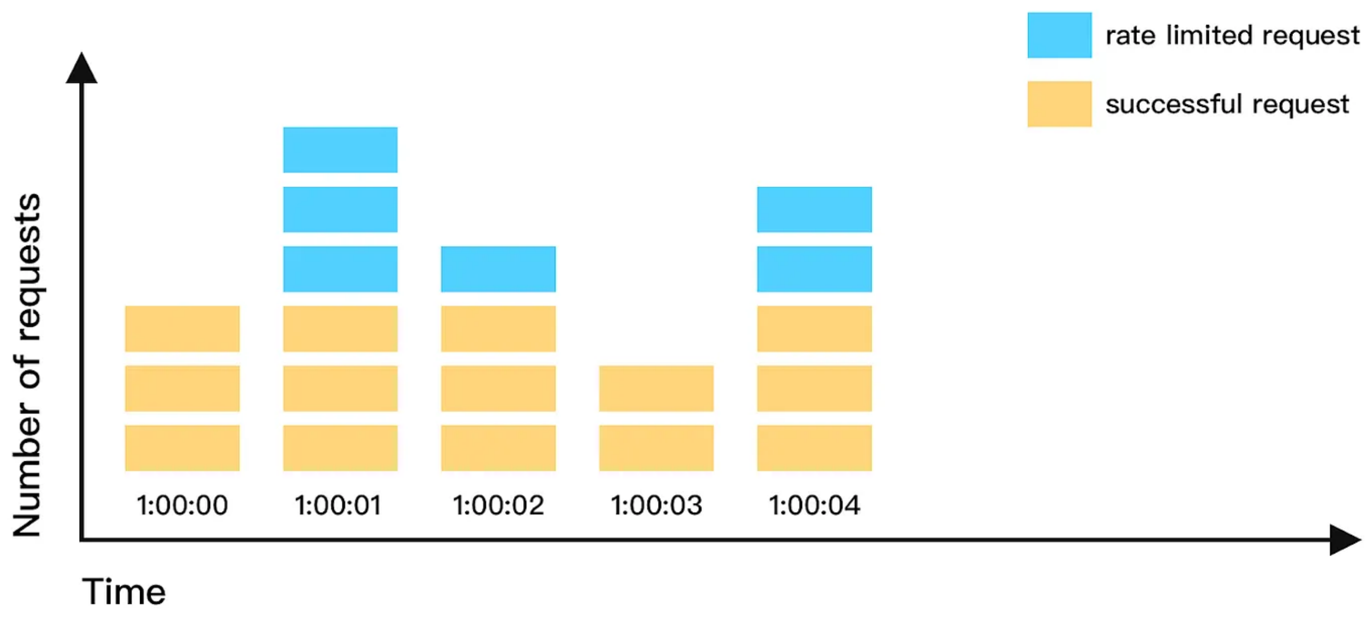
Rate limiting can be implemented using different algorithms. Each of them has distinct pros and cons. We dive into some of the common rate limiting algorithms in this section. We'll cover:

- Fixed Window Counter
- Sliding Window Log

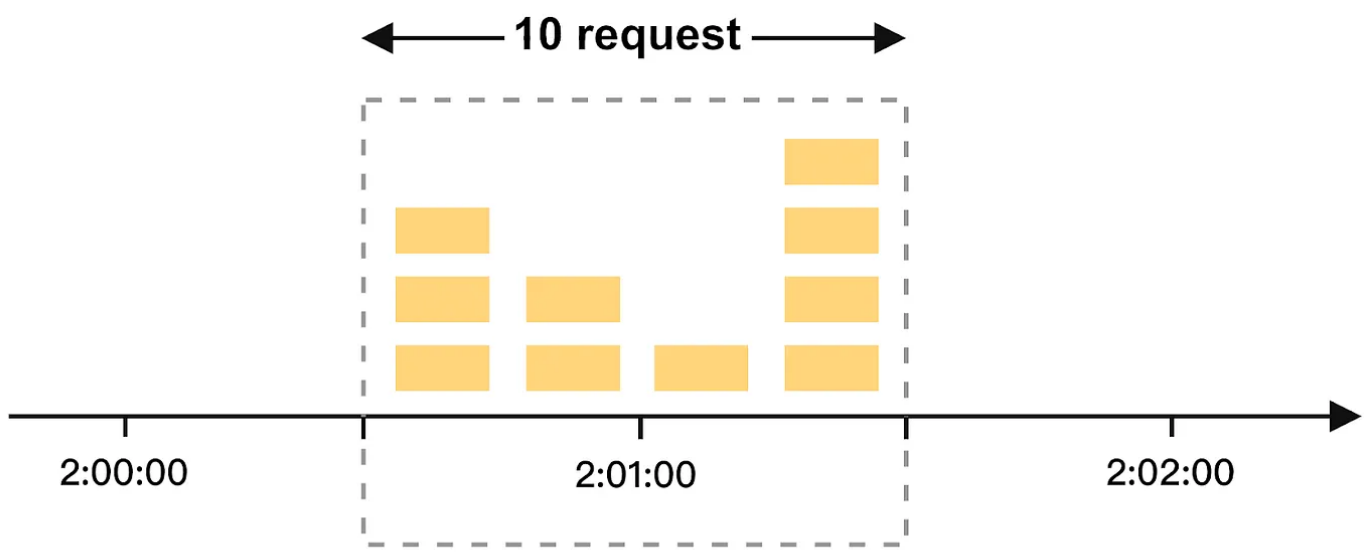
- Sliding Window Counter
- Token Bucket
- Leaky Bucket

Fixed Window Counter

The Fixed Window Counter algorithm divides the timeline into fixed-size time windows and assigns a counter for each window. Each request increments the counter by some value based on the relative cost of the request. Once the counter reaches the threshold, subsequent requests are blocked until the new time window begins.
Let us use a concrete example to see how it works. In this example, the time unit is 1 second and the system allows a maximum of 3 requests per second. In each 1-second time window, if more than 3 requests are received, subsequent requests arriving within the same time window are dropped.

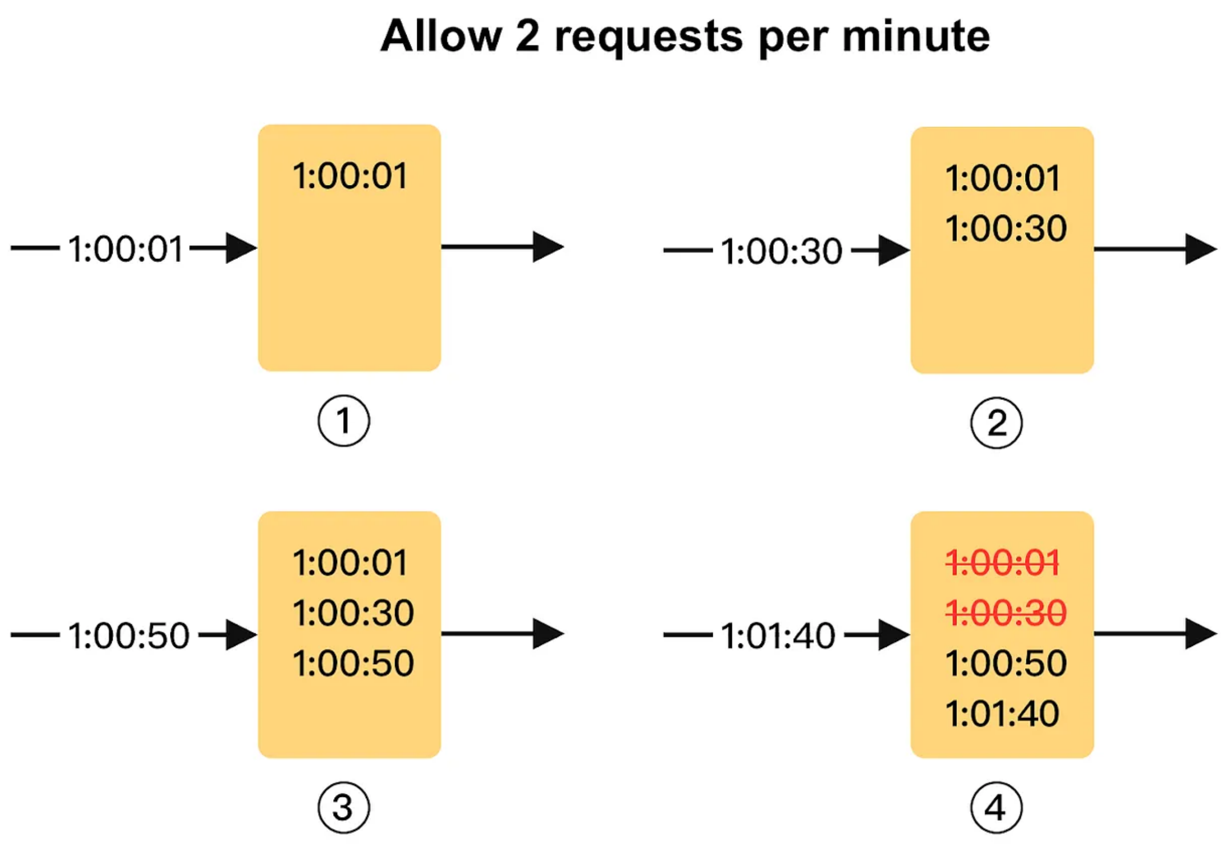


This algorithm is simple to implement. **A major problem with it, however, is that a burst of traffic at the beginning or end of time windows could allow excess requests over the threshold to go through. This causes uneven distribution of requests over time.**
Consider the following example. The rate limiter allows a maximum of 5 requests per minute, and the available quota resets at the top of each minute. There are five requests between 2:00:00 and 2:01:00 and five more requests between 2:01:00 and 2:02:00. For the one-minute window between 2:00:30 and 2:01:30, 10 requests go through. That is twice as many as allowed requests.



Sliding Window Log

The Sliding Window Log algorithm fixes the issue with the Fixed Window Counter algorithm where it allows more requests to slip through at the edges of a time window.
The algorithm keeps track of the timestamps of individual requests in a log. The log is usually kept in a cache, such as sorted sets in Redis. When a new request arrives, the log is checked for requests within the window. The check removes all outdated timestamps older than the start of the current time window from the logs and adds the new request to the log. The new request is allowed if the number of existing requests within the window is below the limit
Let's walk through an example. Here, the rate limiter allows 2 requests per minute. We show human-readable representation of time for better readability.

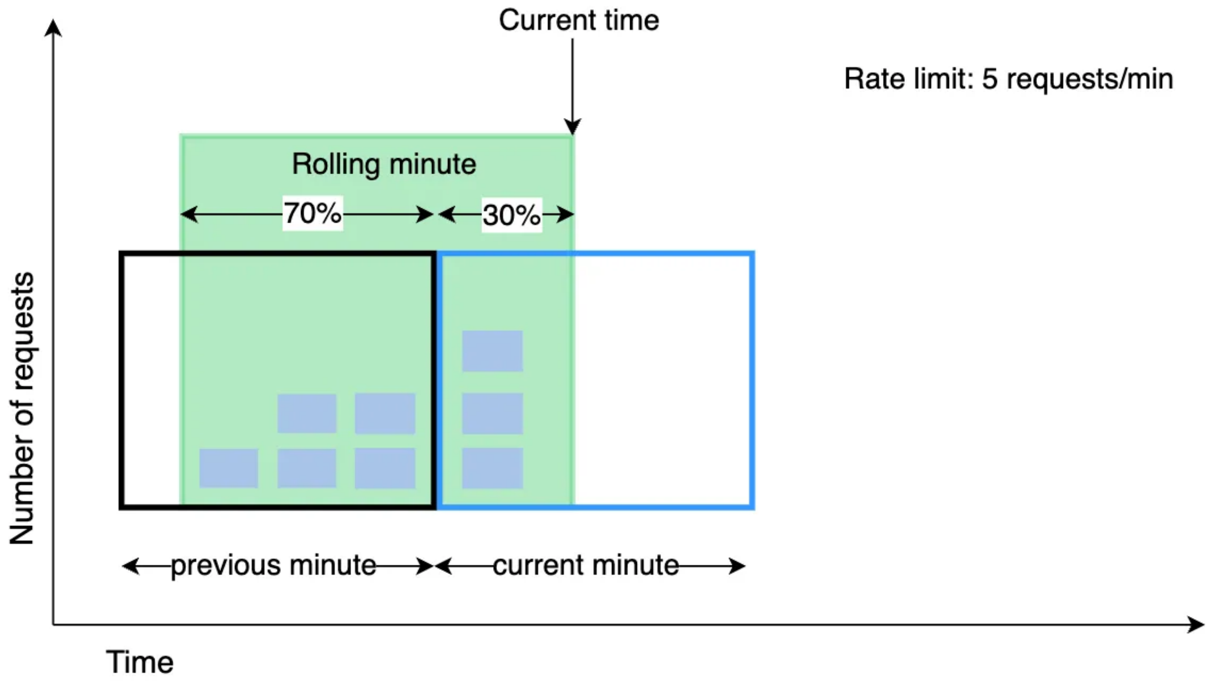


The log is empty when a new request arrives at 1:00:01. The request is allowed.
A new request arrives at 1:00:30, the timestamp is inserted into the log. Now the log size is 2. It is below the threshold of 2. This request is also allowed.
Another request arrives at 1:00:50, and the timestamp is inserted into the log. The new log size is 3, which is larger than the threshold. This request is rejected.
The final request arrives at 1:01:40. Requests in the range [1:00:40,1:01:40) are in the latest time window, but requests sent before 1:00:40 are outdated. Two outdated timestamps, 1:00:01 and 1:00:30, are removed from the log. After clearing the old log entries, the log size is 2, and the request is accepted.
This algorithm ensures a more even distribution of requests. In any rolling window, requests will not exceed the limit.
However, the algorithm consumes a lot of memory. It maintains a log of timestamps, even for requests that are rejected.

Sliding Window Counter

The Sliding Window Counter algorithm is a more efficient variation of the Sliding Window Log algorithm. It is a hybrid that combines the fixed window counter and sliding window log. Instead

of maintaining a log of request timestamps, it calculates the weighted counter for the previous time window. When a new request arrives, the counter is adjusted based on the weight, and the request is allowed if the total is below the limit.
Let’s walk through a concrete example. In this example, the rate limiter allows a maximum of 7 requests per minute. There are 5 requests in the previous minute and 3 in the current minute. For a new request that arrives at 30% (18 seconds) into the current minute, the number of requests in the rolling window is calculated using the following formula:
Requests in current window + requests in the previous window * overlap percentage of the rolling window and previous window.
Using this formula, we get 3 + 5 * 0.7% = 6.5 requests. Depending on the use case, the number can either be rounded up or down. In our example, it is rounded down to 6.

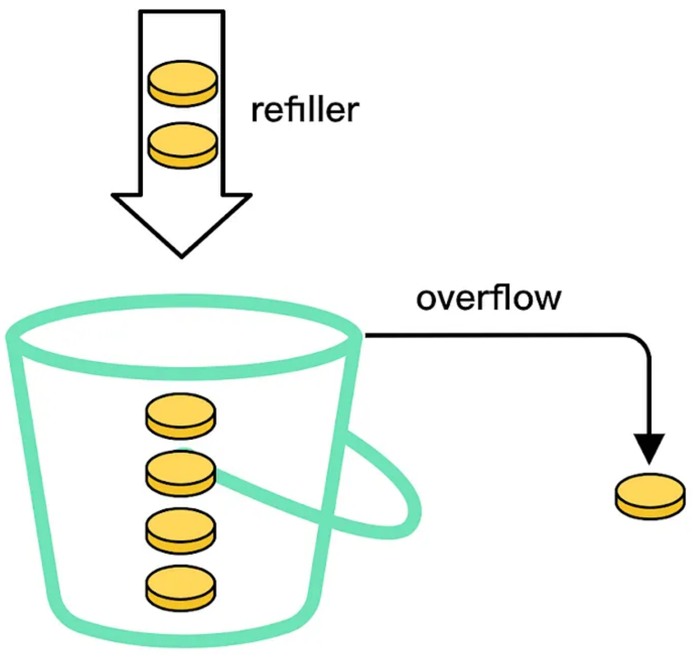


Since the rate limiter in this example allows 7 requests per minute, the current request can go through.
There is another way to implement this algorithm that is more complicated. Instead of computing a weighted counter for the previous window, it uses a counter for each time slot within the window. We will not discuss this other implementation here.

The Sliding Window Counter algorithm has its pros and cons. It smooths out spikes in traffic because the limit includes the average rate of the previous window in the calculation. It also reduces storage and processing requirements compared to the Sliding Window Log algorithm. However, it may still allow bursts of requests to slip through. It is an approximation of the actual rate because it assumes requests in the previous window are evenly distributed. This problem may not be as bad as it seems. According to experiments done by Cloudflare, only 0.003% of requests are incorrectly allowed or rate limited among 400 million requests.

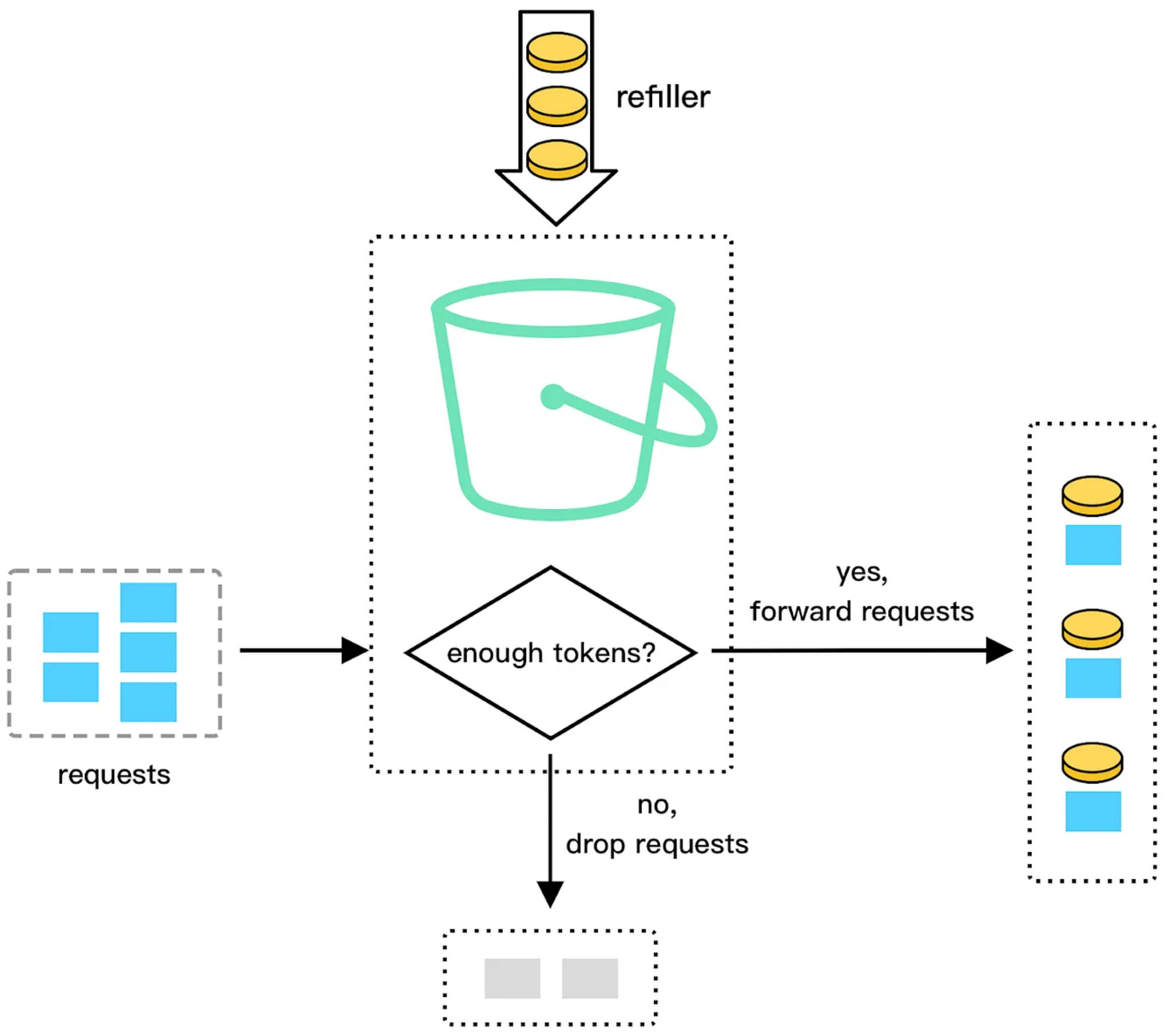
Token Bucket

The Token Bucket algorithm is widely used for rate limiting. It is simple, well understood and commonly used by large tech companies. Both Amazon and Stripe use this algorithm to throttle their API requests.
The Token Bucket algorithm uses a "bucket" to hold tokens. The tokens represent the allowed number of requests. The bucket is initially filled with tokens, and tokens are added at a fixed rate over time. When a request arrives, it consumes a token from the bucket, and the request is allowed if there are enough tokens.
Let’s walk through an example. Here, 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.

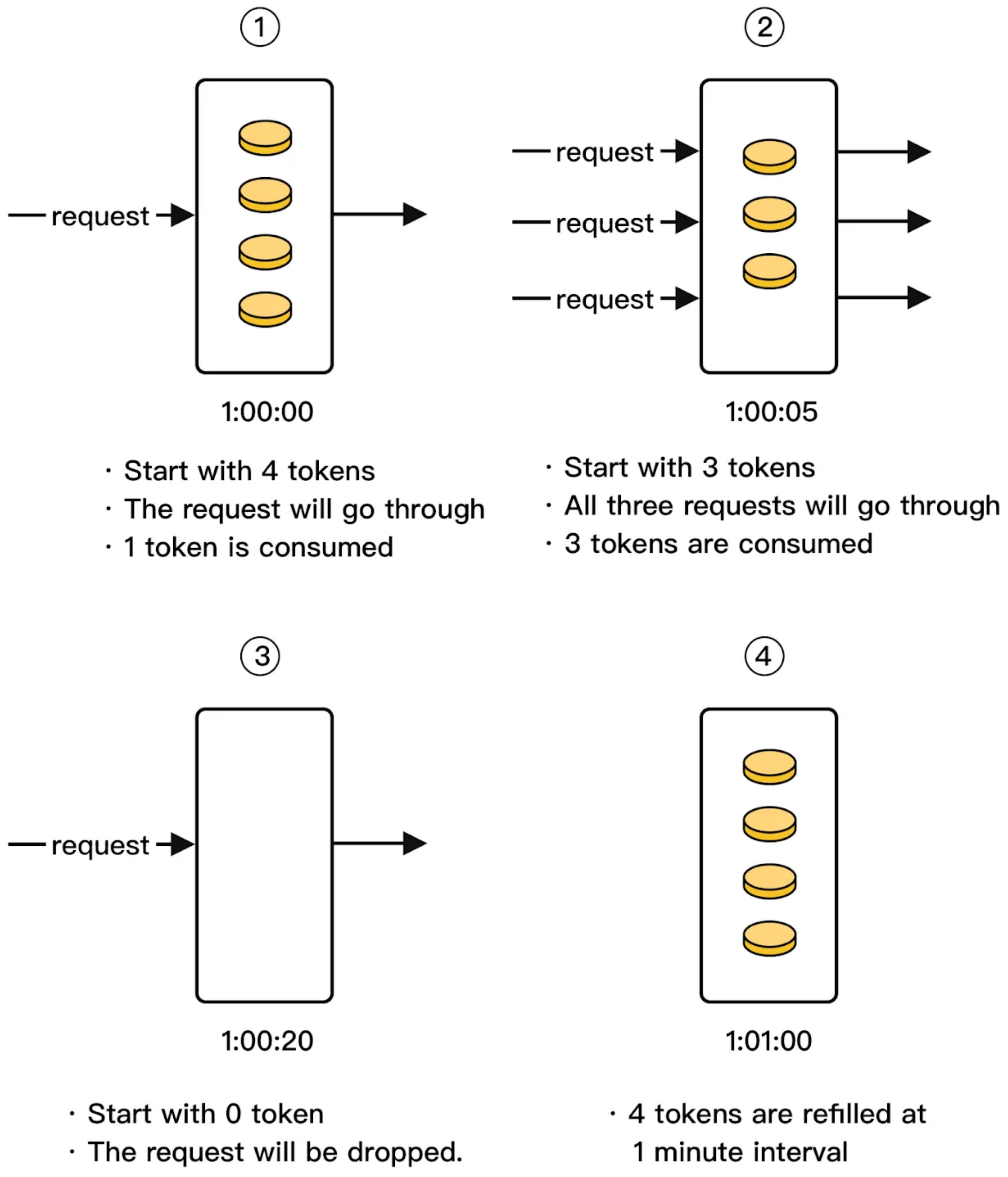


Each request consumes one token. When a request arrives, we check if there are enough tokens in the bucket.

The diagram below explains how it works.
If there are enough tokens, we take one token out for each request, and the request goes through. If there are not enough tokens, the request is dropped.



The following diagram illustrates how token consumption, refill, and rate limiting logic work. In this example, the token bucket size is 4, and the refill rate is 4 per 1 minute.



The token bucket algorithm takes two parameters:

- Bucket size: the maximum number of tokens allowed in the bucket
- Refill rate: number of tokens put into the bucket every second

How many buckets do we need? This depends on the rate-limiting rules. Here are a few examples.

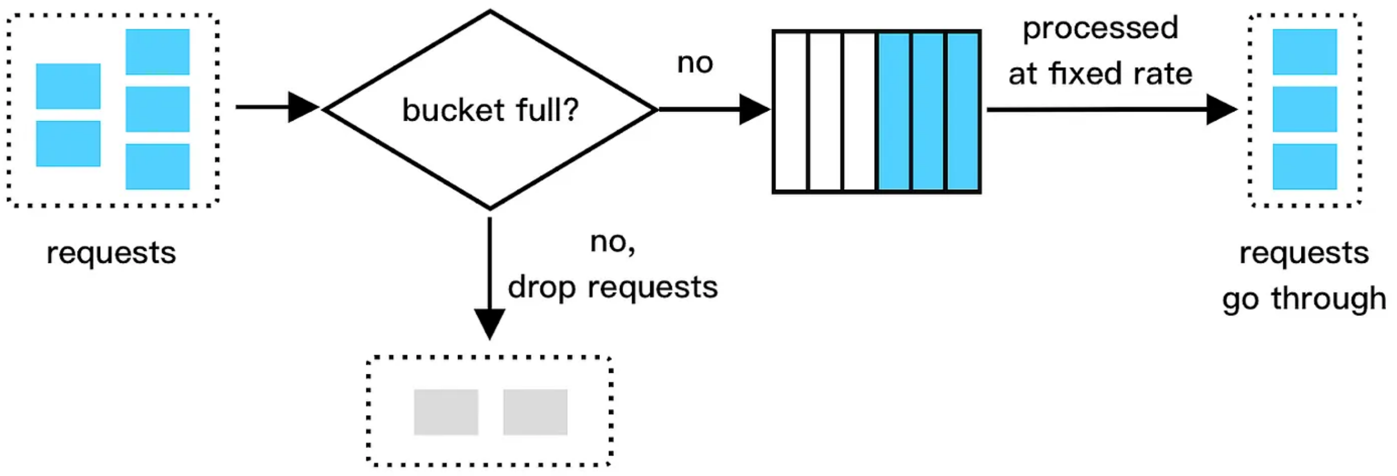
- It is usually necessary to have different buckets for different API endpoints. For instance, if a user is allowed to make 1 post per second, add 150 friends per day, and like 5 posts per second, 3 buckets are required for each user.
- If we need to throttle requests based on IP addresses, each IP address requires a bucket.
- If the system allows a maximum of 10,000 requests per second, it makes sense to have a global bucket shared by all requests.

This algorithm allows for a smooth distribution of requests and can handle bursts of requests up to the bucket's capacity. It is memory efficient and relatively easy to implement.

Leaky Bucket

The Leaky Bucket algorithm also uses a "bucket" metaphor but processes requests differently. Requests enter the bucket and are processed at a fixed rate, simulating a "leak" in the bucket. If the bucket becomes full, new requests are discarded until there is space available. 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.



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 requests per second.

Shopify uses leaky buckets for rate-limiting.

This algorithm is memory efficient given the limited queue size. Requests are processed at a fixed rate. It smooths out request bursts and enforces a consistent rate of processing. It is suitable for use cases where a stable outflow rate is required.
However, a burst of requests would fill up the queue with old requests, and if they are not processed in time, recent requests will be rate limit. It may result in longer waiting times for requests during high-traffic periods.
Each of these rate limiting algorithms has its strengths and weaknesses. The choice of the appropriate algorithm depends on the specific requirements of the system and its desired behavior under various conditions.