Build a High-Performance Rate Limiter

Problem Statement:

You need to implement a **distributed rate limiter** for an API gateway. The rate limiter should limit the number of requests per second per user across multiple instances of the service.

The following are the requirements:

1. **Concurrency**: The system should handle multiple requests concurrently from different users.
2. **Distributed**: The rate limiter needs to work across multiple instances of the service, so rate limits are respected globally (i.e., not just on a single instance of the rate limiter).
3. **Efficiency**: The solution should be memory-efficient and minimize locking overhead. Use Go's concurrency primitives effectively to handle high traffic.
4. **Customizable**: Support dynamic configuration of rate limits for each user (e.g., some users may have a rate limit of 100 requests per second, others 1000 requests per second).
5. **Precision**: Implement a sliding window algorithm to ensure fairness and precision in rate limiting.
6. **Persistence**: In case of system failure, the rate limiter should be able to recover from persistent storage without losing track of request counts for the current time window.

Requirements:

- **Rate limit**: Limit each user to a certain number of configurable requests per second.
- **Distributed environment**: Ensure rate limits are respected across multiple instances of the API gateway.
- **Concurrency**: Handle thousands of requests per second in a concurrent and non-blocking way.
- **Storage**: Use Redis to maintain a global state for rate limits across distributed instances.
- **Sliding window**: Implement a sliding window algorithm to prevent burst traffic from exploiting the rate limits.

Functional Details:

- Assume each request contains a user ID.
- You must design a function `RateLimit(userId string, limit int) bool`, where `limit` is the maximum number of requests allowed per second. This function should return `true` if the request is allowed, and `false` if it is rejected due to the rate limit.
- For each instance of your rate limiter, you should communicate with Redis to ensure global consistency of the rate limit for each user.

- Add comments and documentation for important decisions, especially related to architecture, optimization, and any trade-offs made.

Bonus:

- Add unit tests and benchmarks for your implementation.
- Include documentation on how your solution could scale further or how you'd handle any bottlenecks.
- If time permits, include logic to implement a "leaky bucket" algorithm instead of or alongside the sliding window.

Deliverables:

- A Go implementation of the rate limiter.
- Unit tests for various scenarios (normal usage, edge cases like exceeding limits, and distributed environment behavior).
- A README explaining your approach, architecture, and any assumptions you made.