Design Decisions & Rationale

This document provides detailed answers to the technical challenges of the Rate Limiting Library, focusing on scalability, reliability, and framework-agnosticism.

1. Algorithm Implementation

Token Bucket

- **Request Fulfillment:** Strictly **Binary**. Requests requiring more tokens than available are rejected immediately. Partial fulfillment introduces unpredictable client behavior.
- **Refill Strategy: Continuous (Lazy Calculation)**. Tokens are recalculated at the moment of a request using: tokens = min(capacity, current + elapsed * rate). This avoids background threads and scales linearly with request volume.
- **Initial State:** Buckets start **Full** to allow immediate burst capacity upon service startup.

Sliding Window Counter

- **Granularity:** Configurable from **Seconds to Hours**.
- **Precision:** Uses **Two Windows** (Current and Previous) for a Weighted Moving Average. This provides $O(1)$ memory complexity per user regardless of traffic volume.
- **Memory Safety:** Minimum time unit is 1 second to prevent sub-millisecond keys from exhausting storage memory.

2. Storage Provider SPI

Atomic Operations

- **Redis Strategy:** Use **Lua Scripts**. This ensures atomicity for "Read-Calculate-Write" operations without the overhead of distributed locks or Redis Streams.
- **Management:** Core library provides default scripts; SPI allows overrides for specialized environments (e.g., RedisRaft).
- **Failure Handling:** If a non-Redis provider fails a lock acquisition, the library triggers the **Resilience Policy** (L2 Fallback or Fail-Open).

Data Management

- **Serialization:** Provider-specific. Local storage uses POJOs; Redis uses optimized strings/hashes. Avoid JSON in the hot path.
- **Cleanup (TTL):** Every entry must have a Time-To-Live of $2 \times WindowSize$ to ensure automated memory reclamation for inactive users.

3. Multi-Tenancy & Key Resolution

- **Composite Keys:** Supported. KeyResolver generates a single string key often following

the pattern tenant:user:endpoint.
- **Null Handling:** SpEL resolutions resulting in null are mapped to a "global-anonymous" bucket to prevent service crashes.
- **Global Limits:** Achievable via repeatable annotations or a specialized GlobalKeyResolver.

4. Resilience & Failure Handling

L1/L2 Tiered Defense

- **Recovery Strategy:** When L1 (Redis) recovers, **L2 state is discarded**. Synchronizing L2 state back to L1 introduces high complexity and "write storms" that could re-crash the primary storage.
- **Consistency:** L1 is the single source of truth; L2 is best-effort protection.

Circuit Breaker

- **Granularity: Per-Storage-Provider**. If a storage node is slow, it affects all endpoints.
- **Thresholds:** 50% failure rate over a 10s sliding window.
- **Recovery:** Automatic "Half-Open" check after 30 seconds.

5. Annotation & AOP

- **Multi-Limit Support:** Annotations are @Repeatable, allowing developers to define tiered limits (e.g., 10/sec and 1000/hour) on a single method.
- **Inheritance:** Standard AOP rules apply; method-level annotations override class-level defaults.
- **Resolution:** A single @RateLimit annotation supports both static keys and dynamic SpEL expressions for unified design.

6. Distributed Coordination

- **Config Propagation:** Relies on **Polling/File-Watching** (Kubernetes-native). This avoids the "at-least-once" delivery complexities of Redis Pub/Sub for configuration data.
- **Clock Skew:** All time-based calculations use the **Storage Provider's Clock** (e.g., redis.time()) to ensure cluster-wide synchronization.
- **Telemetry:** Built-in support for **OpenTelemetry** spans to track rate-limit decisions across microservice boundaries.

7. Performance & Priorities

- **Overhead Target:** Local overhead < 500µs; Distributed overhead < 2ms (RTT dependent).
- **Implementation Order:**
    1. Core Engine & SPIs.
    2. Redis (L1) & Caffeine (L2) Providers.

3. Spring & Quarkus Adapters (Parallel).
4. Kubernetes ConfigMap Watcher.