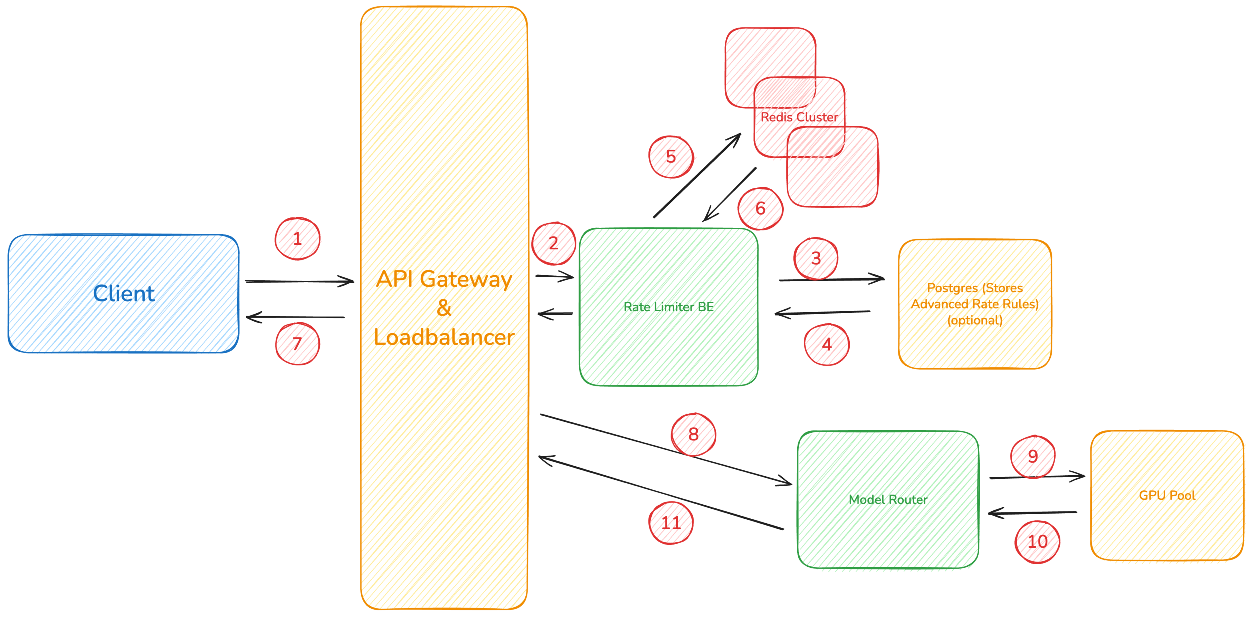
HLD — High-Level Design

2.1 Architecture Overview

Client → API Gateway → Rate Limiter → Model Router → GPU Pool



Client
1
7
API Gateway & Loadbalancer
2
Rate Limiter BE
5
6
Redis Cluster
3
4
Postgres (Stores Advanced Rate Rules) (optional)
8
11
Model Router
9
10
GPU Pool

Purpose of rate limiter in AI serving

- Stops abusive clients from overloading GPUs.
- Enforces tenant-level SLAs.
- Prioritizes internal / premium traffic (QoS tiers).

2.2 Components

Frontend (React)

- Sends requests to backend
- Displays decision

Rate Limiter Backend (FastAPI)

- Implements /rate-limit/check
- Uses Redis Sliding Window

Redis Cluster

- Stores sliding window ZSETs
- May be sharded across multiple nodes

Postgres

- Stores advanced rate policies:
    - per tenant
    - per API key
    - per model tier

2.3 Data Flow

1. UI/API sends rate-limit check
2. Backend builds Redis key
3. Lua script prunes old entries + counts + inserts atomically
4. Backend returns allowed/block
5. Client decides whether to call AI inference API
6. If allowed → request hits GPU pool

2.4 Key Design Choice: Sliding Window Log

Compared to fixed window or token bucket, Sliding Window Log:

- Gives **fairer** enforcement
- Smooths bursts
- Ideal for AI inference where each call may be expensive

2.5 Distributed Design

- Many FastAPI instances → one Redis cluster
- Lua script ensures atomic ops
- Keys hashed across Redis shards
- No cross-shard transactions required

2.6 AI-Specific Extensions

Protecting GPU pools

- Add **model-global** rate limits:
- rl:model:{modelId}:global
- Prevents GPU saturation beyond capacity.

Token-based billing

- Track cost or tokens:
- rl:tokens:{userId}:{modelId}
- Block if exceeding token/hour or cost/day budgets.

QoS tiers

- Tier-based limits:
    - Internal: very high
    - Enterprise: medium
    - Free: strict
- Keys:
    - rl:qos:{tier}:{modelId}

2.6 For very high daily active user count and throughput

1. We might need to shard Redis due to two things:

1. **Memory** – how many timestamps we store
2. **Throughput** – how many ops/sec we need

Because we're using a **Sliding Window Log** with:

- limit = 100 requests/hour (per key)
- each request = 1 ZSET entry
- old entries are purged + TTL removes idle keys
  → **each active key is bounded to ~100 entries**

Rough memory math

Approximate sizing:

- Assume ~100 bytes per ZSET entry (timestamp, metadata, Redis overhead).
- For 100 requests/hour per (user, model):

Per key:

- 100 entries × 100 bytes ≈ 10 KB

Now say we have 1,000,000 active (user, model) pairs:

- 1,000,000 keys × 10 KB ≈ 10,000,000 KB ≈ 10 GB

So:

- **100k active keys** → ~1 GB
- **1M active keys** → ~10 GB

On a typical 8–16 GB Redis instance, **1M+ active keys** starts to be tight (plus overhead for other data), and sharding / clustering becomes attractive.

2. Throughput considerations

Each rate-limit check does:

- ZREMRANGEBYSCORE
- ZCARD
- ZADD
- EXPIRE

Wrapped in a WATCH/MULTI/EXEC pipeline.

A single good Redis node can often handle **100k+ ops/sec**, but that depends on instance size, network, and whether other workloads are sharing it.

Rule of thumb:

- **A few thousand QPS** → single Redis node is totally fine.
- **Tens of thousands QPS sustained**, plus other workloads → **we need to start planning for Redis Cluster or managed sharding**.

3. The current design already supports sharding

The current key design already supports sharding later:

rl:user:{userId}:model:{modelId}
rl:tenant:{tenantId}:model:{modelId}
rl:apikey:{apiKey}:model:{modelId}
...

Redis Cluster will automatically hash these keys across slots/nodes; you do **not** need to change the key format later.

4. What to do when we *do* need sharding

When we grow, we have a few options:

Option A – Redis Cluster (or AWS ElastiCache / Azure Cache)

- Enable Redis Cluster or use managed service with cluster mode.
- Point our client (redis-py) to the cluster endpoints.
- Because each rate-limit check touches **one key**, we don't need cross-slot transactions; each request lives on one shard.

Your current logic still works:

```
key = f"rl:user:{userId}:model:{modelId}"
rate_limiter.check_and_consume(key, window, limit)
```

Just with a **cluster client** instead of a single-node client.

Option B – Separate Redis for rate limiting

Sometimes teams dedicate a separate Redis cluster just for rate limiting, so:

- Rate limiter is isolated from app/session/cache workloads.
- Tuning (maxmemory, eviction policy, etc.) is specific to rate data.

5. Tweaks if we anticipate huge scale

If we expect **tens of millions** of active users or more, we can further reduce pressure:

1. **Use coarser buckets** instead of per-request logs
    - Store counts per minute in a HASH or simple key.
    - Approximate sliding window instead of exact log.
2. **Smaller windows or lower limits**
    - Fewer timestamps per key.
3. **Per-tenant sampling** or **hierarchical limits**
    - Strong limits per tenant; looser per user inside tenant.