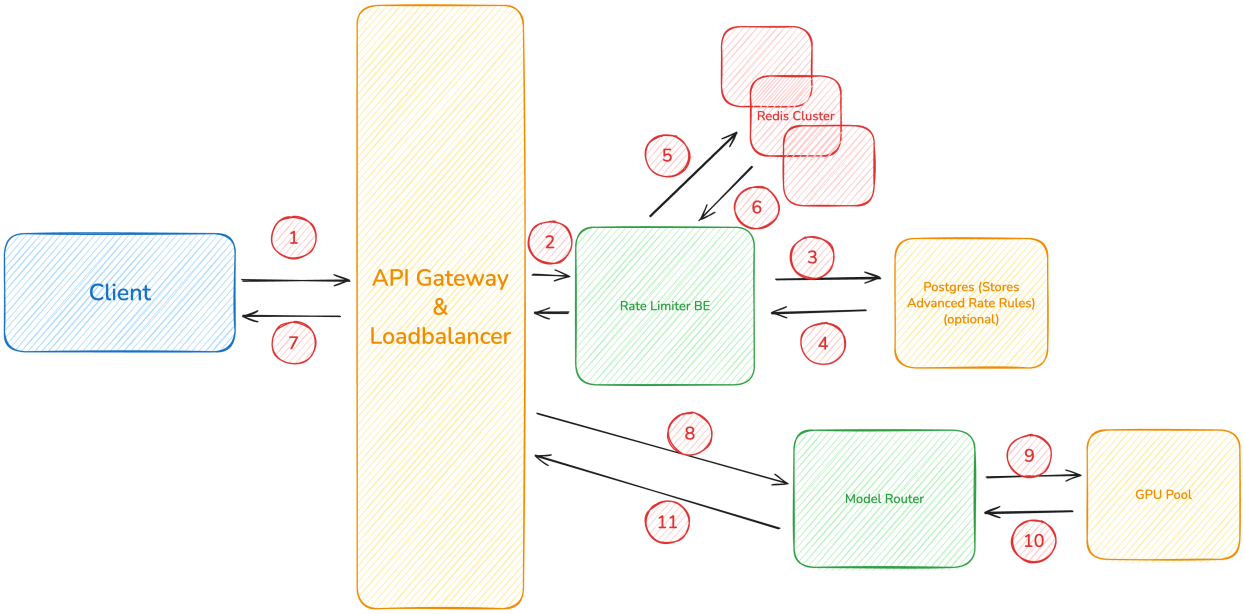
2.1 Architecture Overview

Client → API Gateway → Rate Limiter → Model Router → 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}