1.1 Purpose

Design and implement a **distributed, Sliding Window Log based rate limiter** for AI model serving.
This system prevents GPU overload, enforces fair usage, and supports tenant-based, API key-based, model-tier-based rate decisions.

The solution includes:

- A standalone backend rate limiter in **Python (FastAPI)**
- A **Redis-based** distributed Sliding Window Log
- A simple **React** UI to demonstrate its behavior

1.2 Scope

In-scope

- allow(userId, modelId) conceptual function
- REST API wrapper: POST /rate-limit/check
- Default rate limit: **100 requests/hour per (userId + modelId)**
- Sliding Window Log algorithm
- Distributed concurrency correctness using Redis + Lua
- React demonstration interface
- Extensible rate policy framework

Out-of-scope (for now)

− Authentication, payments, multi-region replication
− Persistent multi-tenant UI
− Token billing engine (designed but not implemented)

1.3 Actors

- **Client Applications**
  Services calling AI inference endpoints.
- **Rate Limiter Backend**
  FastAPI app that enforces limits via Redis.
- **Redis**
  Shared distributed store used for sliding window logs.
- **React Demo User**
  Demonstrates the system visually.
- **Redis**
  Central store for sliding window logs.
- **(Optional) Admin Users**
  Would manage rate limit policies if DB-backed policies are enabled.

1.4 User Stories

1. As a client, I want to know if I can call a model without violating rate limits.
2. As a platform owner, I want to prevent overloaded GPU pools.
3. As a tenant admin, I want tenant-specific rate limits.
4. As a demo user, I want to visualize the rate limiter in real time.

1.5 Functional Requirements

FR1: API

- Conceptual signature:

  bool allow(userId, modelId)

- REST endpoint wrapper:

  POST /rate-limit/check
  → { allowed, limit, count, windowSeconds }

FR2: Base Rule

- 100 requests per moving 1-hour window per (userId, modelId).

FR3: Extensions

- Per tenant limits
- Per API key limits
- Per model tier limits (e.g., GPT-4 stricter than small models)

FR4: Sliding Window Log

- Remove old entries
- Count within window
- Insert new timestamp

FR5: Distributed Guarantees

- Uses Redis + Lua script to ensure atomicity under concurrency.

FR6: React Demo

- Fields: userId, modelId
- Shows ALLOWED / BLOCKED

1.6 Non-Functional Requirements

- Latency: < 5 ms for rate check
- Scale: thousands of requests per second
- Distributed: lock-free, atomic Lua script
- Memory bounded via TTL + ZREMRANGEBYSCORE
- Protection of GPU pool capacity

1.7 Assumptions

- Clocks of instances reasonably accurate
- Redis accessible to all instances
- FastAPI stateless