Assignment: Implement a Rate Limiter for an API Service

Objective:
Build a rate limiter for an API service that enforces different request limits based on user tiers. The rate limiter should ensure that users cannot exceed a specified number of requests within a given time frame. The implementation should handle concurrency and efficiently track user request rates.

1. Data Collection API

- **Description:** Implement an API that allows users to submit social media posts or mentions for analysis.
- **Endpoints:**
  - **POST** /api/v1/analytics/submit
    - **Request Body:**
      - platform (string): The platform from which the data originates (e.g., Twitter, Facebook).
      - content (string): The text content of the post or mention.
      - timestamp (string): The time the post was made (ISO 8601 format).
    - **Response:**
      - status (string): Confirmation that the data was received successfully.
      - analysis_id (string): A unique identifier for the analysis request.
- **Concurrency Considerations:**
  - The service should be able to handle high concurrency with a large number of POST requests as users submit data.
  - Implement a queueing system (e.g., Redis, RabbitMQ) to handle the processing of submitted data asynchronously.

2. Real-Time Analytics Dashboard

- **Description:** Implement an endpoint that retrieves real-time analytics data for a given user.
- **Endpoints:**
  - **GET** /api/v1/analytics/dashboard
    - **Query Parameters:**
      - user_id (string): The ID of the user requesting the analytics data.
      - platform (string, optional): Filter by social media platform.
      - start_time (string, optional): Filter data starting from this timestamp.
      - end_time (string, optional): Filter data up to this timestamp.
    - **Response:**
      - mentions_count (integer): Number of mentions for the specified period.
      - top_hashtags (array): List of top hashtags used in the posts.
      - sentiment_score (float): Average sentiment score of the posts.

Rate Tiers:

- **Tier 1: Free Tier**
  - **Description:** Basic access with limited usage.
  - **Rate Limit:** 10 requests per minute and 100 requests per hour.
  - **Endpoints Covered:** All API endpoints.
  - **Exceeding Rate Limit:** If the limit is exceeded, the API should return a 429 Too Many Requests status code with a message indicating the rate limit has been exceeded.

- **Tier 2: Standard Tier**
  - **Description:** Access with moderate usage limits.
  - **Rate Limit:** 50 requests per minute and 500 requests per hour.
  - **Endpoints Covered:** All API endpoints.
  - **Exceeding Rate Limit:** Return a 429 Too Many Requests status code with a message indicating the rate limit has been exceeded.
- **Tier 3: Premium Tier**
  - **Description:** High access limits for power users.
  - **Rate Limit:** 200 requests per minute and 2000 requests per hour.
  - **Endpoints Covered:** All API endpoints.
  - **Exceeding Rate Limit:** Return a 429 Too Many Requests status code with a message indicating the rate limit has been exceeded.

Requirements:

1. **Rate Limiting Logic:**
   - Implement a rate-limiting algorithm using a token bucket, sliding window, or any efficient method.
   - The rate limiter should be thread-safe and capable of handling concurrent requests.
   - The rate limiter should track requests based on user identifiers (e.g., API keys or user IDs).
2. **Handling Rate Limits:**

   - When a user exceeds the rate limit, the API should immediately return a 429 Too Many Requests response.
   - The response should include information on when the user can retry the request.

3. **Configuration:**
   - Implement a mechanism to configure rate limits for different tiers dynamically. This could be via configuration files or environment variables.
   - Ensure that the system can easily add or modify tiers without significant changes to the codebase.

4. **Efficiency:**
   - Optimise the solution for low memory usage and high performance.
   - Consider using in-memory data structures (e.g., Redis or in-memory caches) to track the rate limits.

5. **Testing:**
   - Write comprehensive tests to ensure the rate limiter behaves correctly under different scenarios, including edge cases like burst traffic.
   - Test cases should include:
     - Requests within limits.
     - Requests exceeding limits.
     - Handling of concurrent requests.
6. **Documentation:**
   - Provide clear documentation explaining the rate-limiting strategy, configuration options, and how to integrate the rate limiter with the API.
   - Include instructions on how to run the service and any dependencies needed.

Deliverables:

- Source code for the rate limiter and API service.
- A README file with:
    - Explanation of the rate-limiting approach.
    - Instructions to run and configure the service.
    - Examples of how to test the rate limits.
- Test cases demonstrating the correct behaviour of the rate limiter under various conditions.