Boardy AI Semantic Caching Challenge

Background

We're building an AI-powered service that handles a high volume of user queries. To optimize performance and reduce costs, we need to implement a semantic caching system that decreases the number of LLM calls while maintaining response quality.

Challenge Overview

Design and implement a semantic caching system that efficiently handles similar queries without requiring redundant LLM calls. Your solution should demonstrate your ability to:

- Understand and implement semantic similarity concepts
- Create an efficient caching mechanism
- Balance performance, cost, and accuracy tradeoffs
- Build a containerized solution that can be easily tested

Requirements

Functional Requirements

1. Create a semantic caching system that:

    - Identifies when incoming queries are semantically similar to previously seen queries
    - Returns cached responses for semantically similar queries
    - Falls back to LLM calls when no suitable cache match exists
    - Maintains cache efficiency with appropriate eviction policies

2. Implement a REST API server that:

    - Accepts query requests
    - Returns appropriate responses (either from cache or LLM)
    - Provides basic metadata on cache hits/misses

3. Package your solution in a Docker Compose setup that can be started with a single command

Technical Considerations

- Use real embedding models and/or LLM APIs of your choice (OpenAI, Hugging Face, etc.)
- Evaluate different similarity thresholds and their impact on accuracy vs. cost savings
- Implement appropriate persistence for the cache
- Design for reasonable performance under load
- Consider time-sensitivity of queries (e.g., "What's the weather today?" should have different caching behavior than "Who was the first president?")

API Specification

Your API should expose at minimum the following endpoint:

```
None
POST /api/query
```

Request body:

```
None
{
  "query": "What's the weather like in New York today?",
  "forceRefresh": false  // Optional parameter to bypass cache
}
```

Response body:

```
None
{
  "response": "The weather in New York today is sunny with a high of 75°F.",
  "metadata": {
    "source": "cache"  // "cache" or "llm"
  }
}
```

Deliverables

1. A complete codebase with your implementation
2. Docker Compose configuration for running your solution
3. A README with:
    - Setup and running instructions
    - Architecture overview and design decisions
    - Explanation of your semantic similarity approach (including why you chose specific embedding models and/or cross-encoders)
    - Discussion of tradeoffs and potential optimizations
4. A short demo video (5-10 minutes) explaining:
    - Your overall approach
    - Key technical decisions
    - How your solution would scale with increasing traffic
    - Performance characteristics and optimization opportunities

Evaluation Criteria

Your solution will be evaluated on:

- Quality and clarity of your code
- Effectiveness of your semantic caching approach
- System architecture and design decisions
- Performance and scalability considerations
- Thoroughness of your documentation
- Ability to explain technical concepts in your demo video

Business Context

The primary goal of this system is to reduce the cost of LLM API calls while maintaining a high-quality user experience. Consider the following:

- LLM API calls are expensive (typically $0.01-$0.10 per call)
- Users often ask semantically similar questions with different wording
- Some queries are time-sensitive (weather, news, sports scores) while others are evergreen (historical facts, definitions)
- Response time is important for user experience
- The system should scale efficiently as query volume increases

Testing Your Solution

We will test your solution against a variety of query patterns, including:

- Exact duplicate queries
- Semantically similar queries with different wording
- Completely unrelated queries
- Time-sensitive vs. evergreen queries
- Queries of varying complexity and length
- Queries in different languages or with special characters
- Rapid succession of queries (load testing)

We'll evaluate both the accuracy of your cache hits/misses and the overall system performance under various loads.

Extension Areas (Optional)

To showcase your skills further, consider implementing some of these extensions:

1. **Robust Logging & Monitoring**

    - Structured logging with different severity levels
    - Metrics collection for cache performance (hit rate, latency, etc.)
    - Visualization of cache performance over time

2. **Performance Optimization**

    - Benchmarks comparing different embedding models
    - Optimized similarity calculation techniques
    - Parallel processing for high throughput

3. **Advanced Caching Strategies**

    - Time-based cache invalidation for time-sensitive queries
    - Topic-based cache partitioning
    - Hierarchical caching with different similarity thresholds

4. **Load Testing & Resilience**

    - Include load testing scripts and results
    - Circuit breakers for external dependencies
    - Graceful degradation under high load

5. **Enhanced API Features**

    - Endpoints for cache statistics and management
    - Query categorization or tagging
    - Confidence scores for cache matches

Submission Guidelines

Submit your solution as follows:

1. Create a private GitHub repository with your implementation

2. Invite the following GitHub accounts to your repository:

    - `owengretzinger`

3. Include a `.env.example` file in your repository with all required environment variables (but no actual credentials). Make sure the variables are named in an interpretable way so we can load our own for testing purposes.

4. Ensure your Docker Compose setup works on a standard environment with a single command and exposes the server at `localhost:3000`

Complete the challenge within one week of receiving it.

Good luck! We're excited to see your creative approach to this problem.