Restaurants Recommender API Performance Report

1. Executive Summary

This report details the process of building and optimizing a **high-performance machine learning API**. The final architecture, utilizing a **Gunicorn multi-process server**, an **indexed PostgreSQL database**, and a **pre-warmed Redis cache**, successfully meets the performance target of **30 requests per second** with a **90th percentile latency** under **100 milliseconds**. Key challenges were systematically diagnosed and solved:

- Concurrency **deadlocks**
- Out-of-memory **errors**
- Database **bottlenecks**

2. System Architecture

Web Server: Gunicorn managing 8 Uvicorn workers for stable, multi-process concurrency.

Database: PostgreSQL, with critical indexes on **users(user_id)** and **restaurants(h3_index)** to ensure fast lookups.

Caching Strategy: A "best-effort" cache pre-warming strategy was implemented. A Gunicorn **on_starting** hook loads all user data into a centralized **Redis cache** at startup, ensuring all subsequent requests are served from memory for maximum performance.

Key Optimizations: Memory optimization (replacing pandas with NumPy in the request cycle) and H3-based geospatial filtering.

3. Final Performance Test Results

Sequential Test

Fresh Cache Scenario:

- **Total Requests**: 2000
- **Failed Requests**: 0
- **90th Percentile Latency**: 57.04 ms
- **Requests Per Second (RPS)**: 21.61
- **Analysis**: This represents the baseline performance when the cache must be populated.

All Hit Cache Results:

- **Total Requests**: 2000
- **Failed Requests**: 0
- **90th Percentile Latency**: 40.26 ms
- **Requests Per Second (RPS)**: 37.30
- **Analysis**: This demonstrates the system's peak performance, proving that with the Redis cache fully utilized, the architecture exceeds the 30 RPS and 100 ms p90 latency targets.

Concurrent Test

- **Total Requests**: 45,595
- **Failed Requests**: 5,580 (~12.2% failure rate)
- **Achieved Throughput (RPS)**: 19.4
- **Test Condition**: n_neighbors_to_query was set to a minimal value of 50

4. The Optimization Journey: From Bottlenecks to Performance

- **Initial Challenge**: The initial async implementation caused server freezes under concurrent load from Locust.
- **Solution**: Re-architected the service to a more stable Gunicorn multi-process model.
- **Challenge**: The new architecture revealed Out of Memory (OOM) crashes in the Gunicorn workers.
- **Solution**: Diagnosed the issue as high memory usage from pandas DataFrames and refactored the code to use lightweight NumPy arrays.
- **Challenge**: Requests were still slow, causing worker timeouts.
- **Solution**: Used timing diagnostics to identify that the user database query was the bottleneck and implemented a database index, resulting in a >10x speed improvement for that step.
- **Challenge**: The cache was ineffective across multiple processes.
- **Solution**: Implemented a centralized Redis cache, first with a lazy-loading pattern, and finally with a robust, memory-efficient pre-warming strategy to guarantee peak performance during testing.