Comprehensive Caching Mechanism for MedLib's AI Healthcare Assistant

COM S 402C: Senior Design, Fall 2024

Instructor: Simanta Mitra

TA: Shaiqur Rahman

Client: Eranda Sooriyarachchi

Mohamed Aradaib

Jagger Gourley

Owen Harty

Jason Jiang

Steven Tran

Chanho Yang

Table of Contents

Table of Contents	**2**
Letter from the Client	**3**
Requirements	**4**
Design	**5**
Work Done	**6**
Results Achieved	**7**
Appendix	**8**
Final Report Instructions	**9**

Letter from the Client

TAs to attach - a letter from the client expressing their thoughts/feelings on the team's work.

Placeholder for the letter from the client.

Requirements

Problem Statement: Cache queries from MedLib's AI Healthcare Assistant to reduce expensive API calls to their LLM.

Objectives:

- Query/Response pair storage
    - New queries that do not have a match in the caching system is stored in the ChromaDB database
    - Queries are stored as their embeddings to aid in later semantic retrieval
    - Database parameters are easily modifiable
- Exact query retrieval
    - Retrieve stored response if a received query exactly matches the stored query
- Semantic query retrieval
    - Retrieve stored response if a received query semantically matches the stored query
- API endpoints
    - Expose endpoints that allow the frontend to query the database and retrieve cached responses
    - Expose endpoints that allow manual manipulation of query/response pairs, storage parameters, and semantic matching parameters

Primary Users:

- Healthcare professionals and MedLib team.

Functional Requirements:

- Query caching (exact and semantic).
- Configurable similarity thresholds.
- LRU eviction policy.
- Persistent storage using ChromaDB.

Non-functional Requirements:

- Sub-second response time for cached queries.
- Scalability for handling # of queries per second.
- Compliance with data privacy standards.

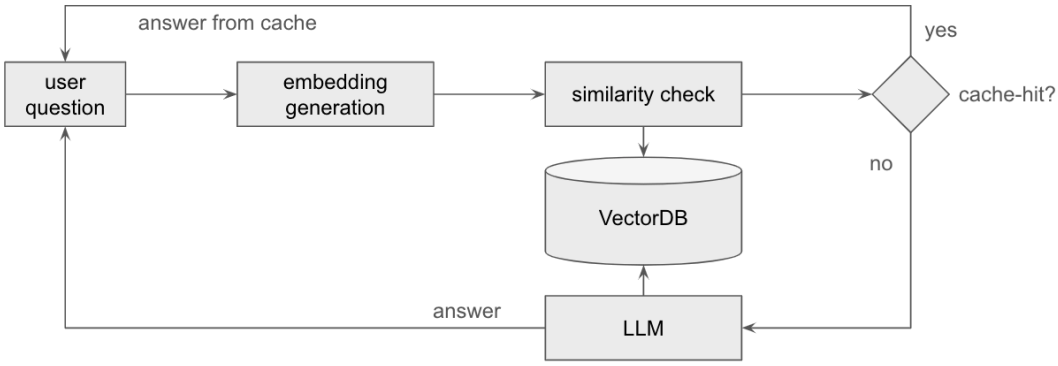
Constraints:

- Avoid caching PII.

Design

System Architecture:

The caching mechanism is designed to optimize the MedLib AI Healthcare Assistant by handling queries before they reach the LLM. The architecture integrates four core components: exact query handling, semantic query handling, cache persistence, and a cache management API. When a query is received, the system first checks for an exact match in the cache. If no exact match is found, a semantic similarity check is performed using SentenceTransformers embeddings. If neither match is found, the query is forwarded to the LLM, and the response is cached for future use. Persistent storage with ChromaDB ensures that cached data survives system reboots.



API Design:

The caching mechanism exposes a comprehensive API to interact with the system. These endpoints enable query processing, cache management, and configurability of parameters such as similarity thresholds and cache size.

- **Query Handling:**
  - *GET /cache/get/{query}*: Retrieves a cached response for a query (if available).
  - *POST /cache/query*: Accepts a query, checks the cache, and either returns a response or calls the LLM.
- **Cache Management:**
  - *POST /cache/add*: Adds a query-response pair to the cache.
  - *POST /cache/remove/{query}*: Removes a specific query-response pair from the cache.
  - *POST /cache/clear*: Clears all cache entries.
- **Threshold and Size Management:**

- *POST /cache/update/threshold/{threshold}*: Updates the semantic similarity threshold.
- *POST /cache/update/size/{size}*: Adjusts the maximum cache size.

Design Decisions:

- **Semantic Similarity Detection:**
  - The SentenceTransformers model (all-MiniLM-L6-v2) was chosen for its balance between performance and speed in generating embeddings for semantic search.
- **Eviction Policy:**
  - Least Recently Used (LRU) eviction ensures efficient memory management by removing the oldest accessed entries when the cache exceeds its capacity.
- **Persistent Storage:**
  - ChromaDB provides an efficient, persistent storage layer for caching query-response pairs, embeddings, and metadata.
- **Configurability:**
  - The cache size and similarity threshold are configurable via exposed API endpoints to allow for dynamic tuning based on system performance and requirements.

Challenges Addressed:

- **Embedding Size and Retrieval Optimization:**
  - Efficient handling of embedding vectors and leveraging cosine similarity searches ensures rapid retrieval of semantically similar queries.
- **Threshold Tuning for Semantic Caching:**
  - The similarity threshold for semantic matches was set to 0.15 initially, balancing retrieval accuracy and performance. This value is adjustable via the API.
- **Scalability:**
  - Using an LRU eviction policy to manage memory consumption. Exposing API endpoints to adjust cache parameters dynamically. The entire application can also be containerized using Docker.

Work Done

Overview:

The team created a functional caching application that is lightweight and easily deployable. The application exposes several endpoints to interact with the key systems of the cache. For the cache, the app has both exact and semantic caching mechanisms, as well as a flow that allows a user to check queries sequentially through each caching system. All of the data is embedded and stored persistently using ChromaDB. The database as methods for efficient data storage using cache eviction systems. The semantic retrieval process is able to identify semantically similar queries with tunable parameters on how sensitive the matching system is. For any data that is stored, we ensure that PII is omitted due to the nature of the use cases for the application. Our endpoints are secured, and our whole application can be deployed anywhere using Docker.

Team Contributions and Learning Outcomes:

Mohamed:

Jagger:

My contributions included implementing the semantic caching mechanism, setting up integration with ChromaDB for persistent storage, and ensuring the proper interaction of these components with the LLM API through a custom client. I initially configured ChromaDB as a persistent backend, implementing methods for efficient data storage and retrieval. Additionally, I collaborated to combine semantic caching with exact caching into a unified CacheManager class and created unit tests for multiple system components. I implemented logic to retrieve semantically similar responses using SentenceTransformers (all-MiniLM-L6-v2) and cosine similarity. I integrated it with ChromaDB for persistent storage of embeddings and metadata. I also created a client for interacting with the LLM API, handling requests and error responses. The technologies I used/learned to achieve these things included Python, ChromaDB, SentenceTransformers, HuggingFace API, and Python's unittest.

Owen:

My contributions included implementing the semantic cache plus the eviction methodology (Least Recently Used). Initially, we utilized a First in First Out eviction policy; however, after further research, we determined that LRU would be more efficient. Next, I worked on several additional features for the cache such as size and threshold modifications and tracking for the cache's hit/miss rate. I observed basic pseudocode for the LRU policy and modified it to function for our data, which consisted of queries, responses, embeddings, and metadata. I also created some unit tests for various system components and sufficient logging to help future clients debug issues in the future. The technologies I utilized and expanded my

expertise in included Python, ChromaDB, SentenceTransformers, the HuggingFace API, FastAPI, and Python's unittest framework.

Jason:

My contributions include designing and implementing the API system, exact caching mechanism, and project infrastructure such as python environment setup and docker. I set up the initial infrastructure of the app, providing instructions on how to set up python, virtual environments, install packages through pip and requirements, and running the app with FastAPI. Towards the end, I set up Docker for the app to be able to deploy on AWS servers for the client. I set up the API endpoints and developed an easily accessible response format. I collaborated with the team to connect their functionality to the API methods. In addition, I set up the routing between the different caches in the API methods. Finally, I set up the exact caching system as the basic form of caching. I integrated the exact cache with the ChromaDB when that was set up.

Steven:

For the project, one of my contributions was setting up the PII masking for the caching system, which is blocking any private information that the user may give the LLM. I used a HuggingFace model to figure out which details of the LLM response needed to be masked. This includes, but is not limited to, names, phone numbers, emails, and more medical specific information such as insurance numbers. I also helped set up the initial exact caching system as well as helping out with persistence through ChromaDB. The technologies that I used for this project are Python, FastAPI, the HuggingFace API, ChromaDB, SentenceTransformers and Python's unittest.

Chanho:

My contribution to the project involved setting up robust unit tests for the Exact Cache and conducting integration tests for API endpoints, ensuring the system's reliability and functionality. Additionally, a GitHub Action CI pipeline was configured, automating the testing process and maintaining code quality throughout development. To enhance security, Auth0 was implemented for centralized authentication management, simplifying the process of securing API endpoints and enabling a scalable, consistent approach to handling user access and permissions. These contributions collectively strengthened the project's foundation, ensuring seamless functionality, streamlined workflows, and improved security.

Results Achieved

Our project delivered a fully functional and scalable caching mechanism for the MedLib AI Healthcare Assistant, meeting all core objectives in terms of performance and usability. The system includes both exact and semantic caching mechanisms, leveraging SentenceTransformers for semantic similarity detection and ChromaDB for persistent storage. We implemented configurable thresholds and eviction policies, providing flexibility for tuning based on system needs. The caching mechanism is accessible through a comprehensive API, facilitating query handling, cache management, and parameter adjustments.

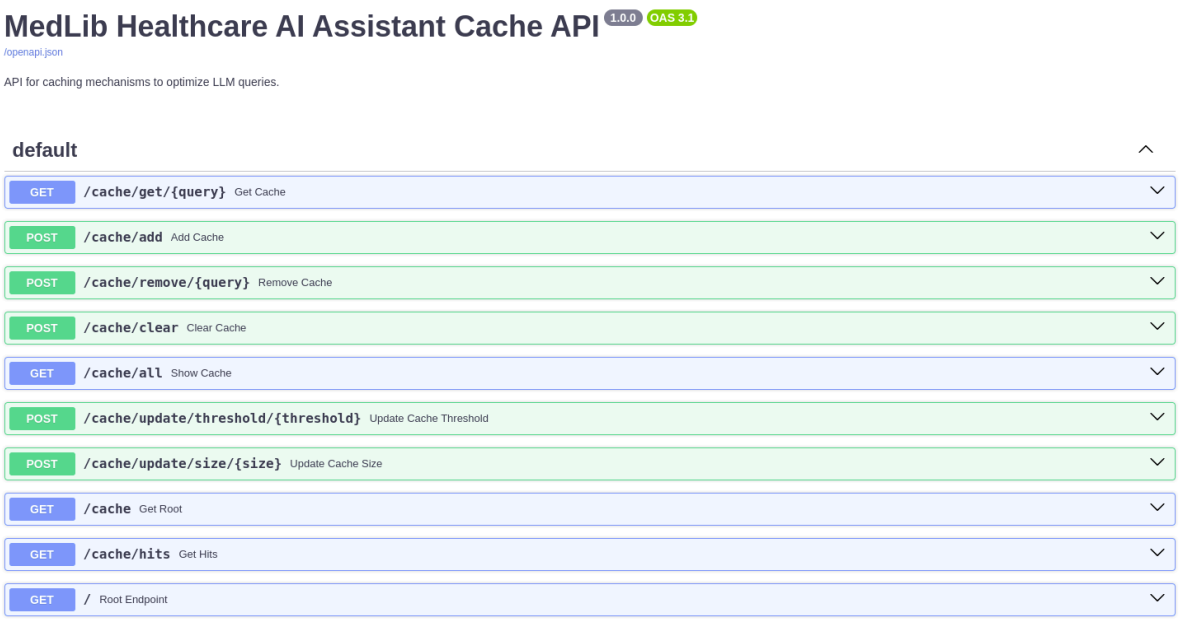
Features Delivered:

- Integration of exact and semantic caching mechanisms.
- Configurable similarity thresholds for semantic matches and dynamic cache size adjustment.
- Persistent storage using ChromaDB to ensure cached data survives system restarts.
- API endpoints for efficient query handling and manual cache management.
- Robust unit tests for key components, ensuring system reliability and performance under various scenarios.

Business Impact:

- Cost Savings: The caching mechanism significantly reduced redundant LLM calls, decreasing operational costs associated with API usage.
- Scalability: The system is designed to handle a high volume of queries and to be easily refactored and containerized, making it capable of future expansion.
- Customization: The system allows clients to customize the size threshold for cached responses, enabling them to tailor the caching mechanism to align with their specific needs and preferences.
- Usability: Comprehensive documentation was provided for all components, including setup instructions, API usage examples, and detailed explanations of caching logic, ensuring ease of adoption and efficient system maintenance for developers.
- Feasibility: The system is user-friendly, with a straightforward structure and extensive logging to facilitate debugging, making it both easy to use and maintain.

Screenshots/Examples:



Appendix

MedLib AI Caching

Group 6:
Mohamed Aradaib, Jagger Gourley, Owen Harty, Jason Jiang, Steven Tran, Chanho Yang

MedLib Healthcare AI Caching System

- MedLib Healthcare AI Assistant aims to provide healthcare professionals with fast, accurate clinical information.
- The project focuses on optimizing performance using caching techniques to reduce redundant inference calls.
- The AI Assistant is powered by a large language model with multimodal capabilities.
- We plan to plan to implement two types of caching-exact and semantic-to improve response times and reduce operational costs.

Problem

Redundant AI inference calls which leads to increased system load and slower response times.

This inefficiency results in higher computational costs and affects user experience due to longer wait times.

We plan to Implement a caching mechanism to store queries, reducing repeated AI calls.

By leveraging exact and semantic caching, the system will enhance performance, reduce load, and provide faster responses to users.

Design and Development Challenges - API Design

- Frontend application is external and is being developed by client
- Goal of our project is to be portable system that can be deployed anywhere
- API will act as a bridge between frontend application and caching system
- FastAPI preferred as one of the fastest and lightweight Python frameworks for developing REST API

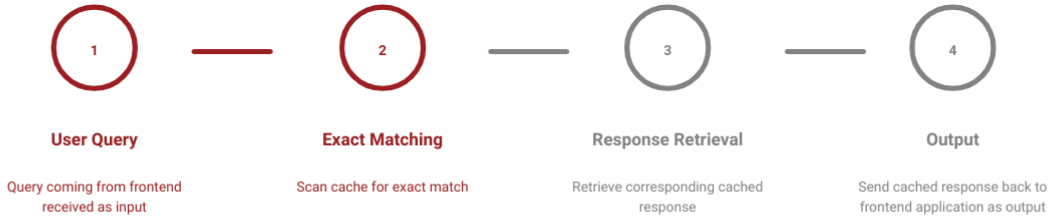
```python
@router.get("/queries/get/{query}")
async def get_diagnosis(query: str):
    response = medical_queries_responses.get(query, "No diagnosis found for this query.")
    return {"query": query, "diagnosis": response}

@router.post("/queries/add")
async def add_query_response(data: QueryResponse):
    query = data.query
    response = data.response
    if query in medical_queries_responses:
        raise HTTPException(status_code=400, detail="Query already exists.")

```

Design and Development Challenges - Exact Matching

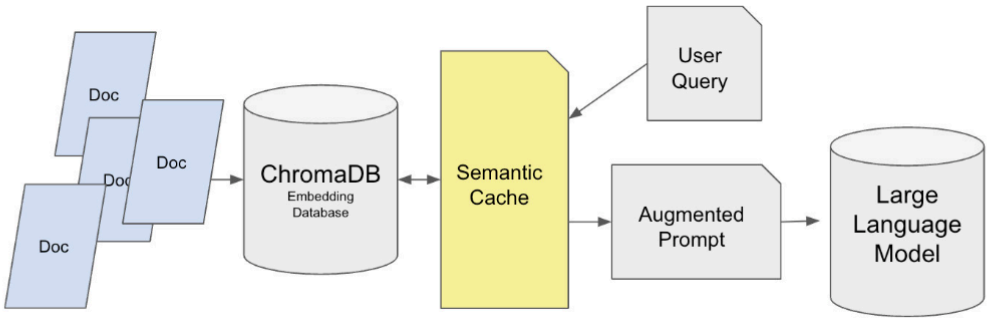
- Most basic form of handling queries and matching
- Substantially less development time than Semantic Caching
- **Important Note:** Cache retrieval only happens when query is **exactly** the same as a previous query
- Result is a simple-form, but extremely limited caching system



Design and Development Challenges

Part One, Semantic Cache:

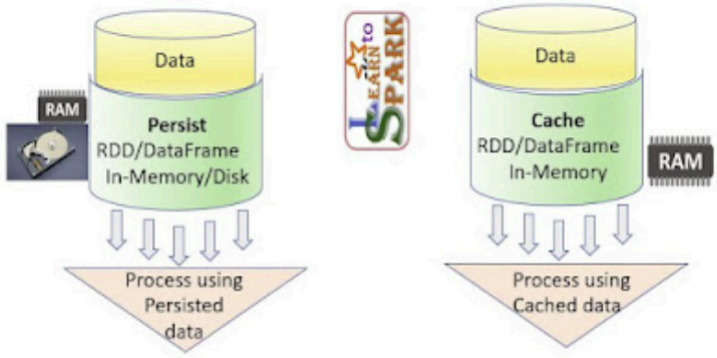
- Use embeddings (Sentence Transformers) to semantically match queries
- Store and retrieve query embeddings along with responses
- Perform similarity checks using cosine similarity or vector search (utilizing Faiss)



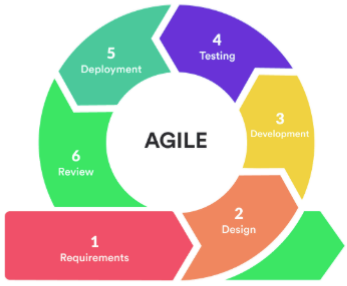
Design and Development Challenges

Part Two, Persistence:

- Option to store the cache persistently (across reboots)
- Integration with Redis, PostgreSQL, or another database



Software Development Practices & Tools





FastAPI

chroma

git + GitHub

SBERT.net

docker

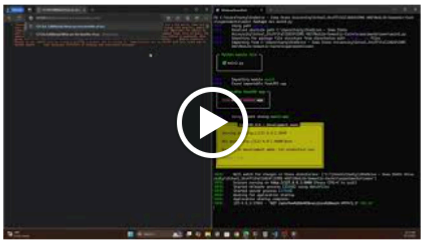
Current Progress & Challenges

- **Exact Cache**
  - Cache new responses
  - Saves cache as a file
- **Semantic Cache**
  - Cache responses
  - Word Embeddings
- **Challenges**
  - Not caching PII
  - Eviction process
  - Persistent memory with ChromaDB
  - Connecting with LLM
  - Manual Changes
  - Logging cache hits and misses

Remaining Work

1. **Exact Cache**
   i. Implement the exact cache architecture
   ii. Test and Validate the cache functionality
2. **Semantic Cache**
   i. Implement the semantic cache architecture
   ii. Test and Validate the cache functionality
3. **FastAPI**
   i. Integrate the exact and semantic cache with FastAPI
   ii. Develop the necessary API endpoints
   iii. Test and Validate the API functionality
4. **Persistence**
   i. Design the data storage schema
   ii. Implement the persistence layer
   iii. Integrate the persistence layer with FastAPI
   iv. Test and Validate the persistence functionality

Demo

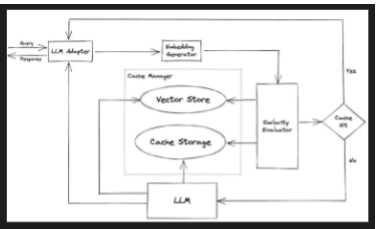


MedLib AI Caching

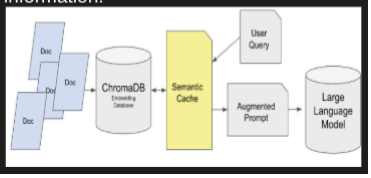
Demo 2

Group 6:
Mohamed Aradaib, Jagger Gourley, Owen Harty, Jason Jiang, Steven Tran, Chanho Yang

Introduction & Project Overview

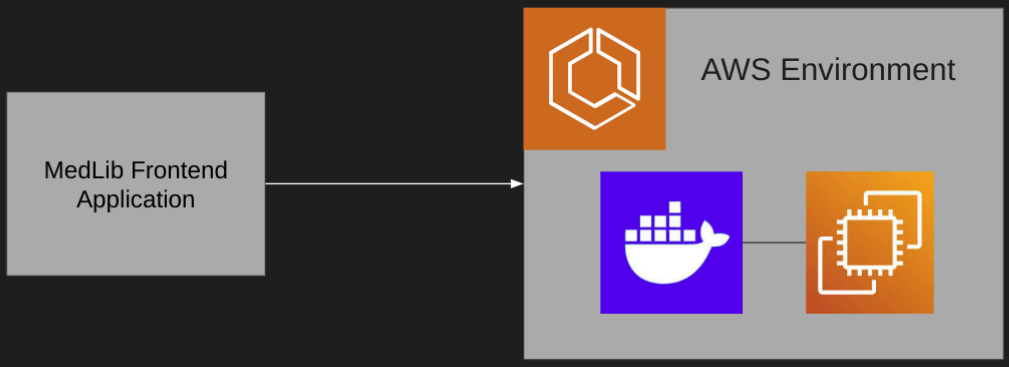


- **Purpose**: Improve response times and reduce operational costs for MedLib's AI Assistant.
- **Objective**: Enhance system efficiency by implementing a caching mechanism to minimize redundant processing.
- **Goal**: Optimize MedLib's AI by reducing repeated processing of similar or identical queries, improving performance and cutting costs.
- **Problem**: Frequent repeated queries cause costly, time-consuming calls to the large language model (LLM).
- **Solution**:
  - **Exact Cache**: Stores identical queries for quick retrieval.
  - **Semantic Cache**: Uses a vector database to identify and store responses for semantically similar queries.
- **Benefits**:
  - **Improved Performance**: Faster response times and reduced system load.
  - **Scalability and Security**: Built to scale with user demand, compliant with GDPR and HIPAA.
- **Impact**: Provides healthcare providers with faster, cost-effective access to reliable medical information.



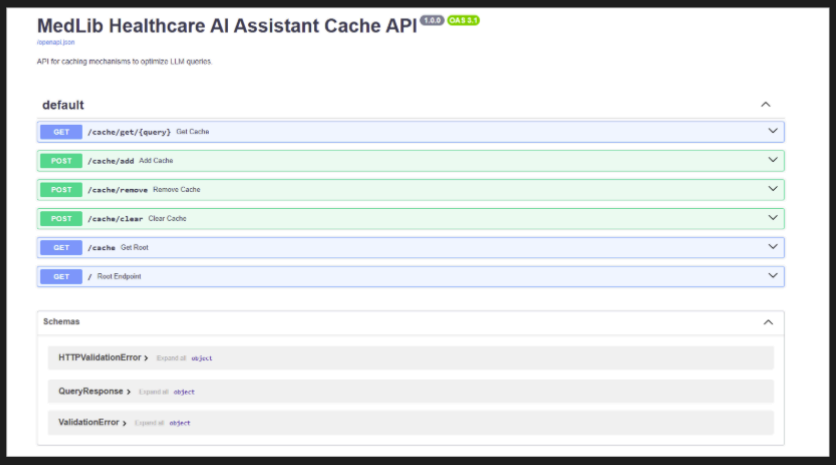
App Services and Integration with Client

- Providing a backend service for MedLib through API endpoints
- Easy containerization and deployment with Docker



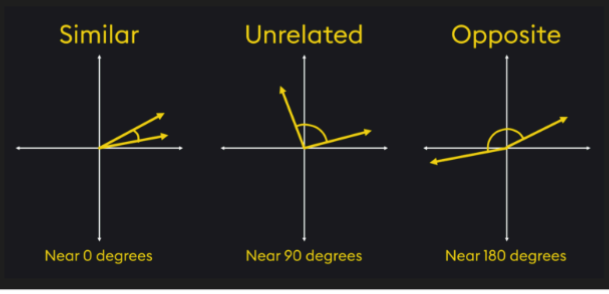
API Design

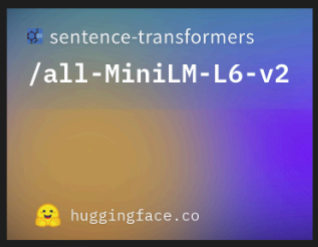
- FastAPI to design REST APIs
- Tailor responses to client
- Manual database management

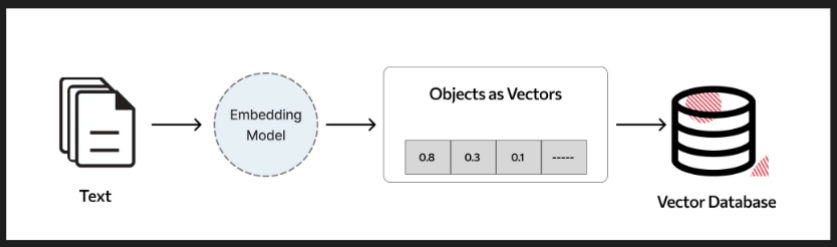


Exact and Semantic Caching Tools

- Exact vs Semantic Matches
- Embeddings & cosine distance
- Chroma DB for efficiency (handles vector data)
- Persistence: Stores queries, responses, embeddings



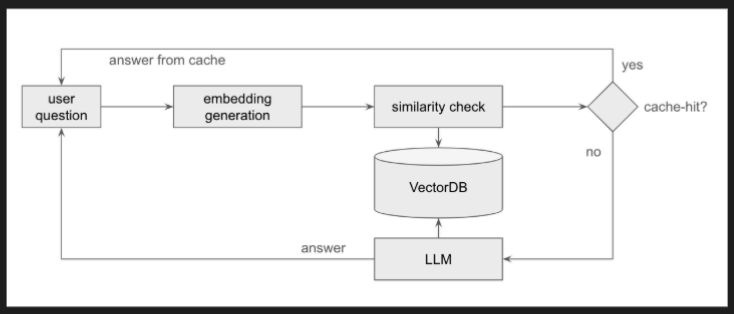




Exact and Semantic Caching Flow

- Exact Match: Check by query ID
- Semantic Match: Compare embeddings (cosine distance)
- Cache Miss: Query LLM, store result
- Model & Threshold considerations





$$\text{Cosine Distance}(\mathbf{A},\mathbf{B}) = 1 - \frac{\mathbf{A}\cdot\mathbf{B}}{\|\mathbf{A}\|\,\|\mathbf{B}\|}$$

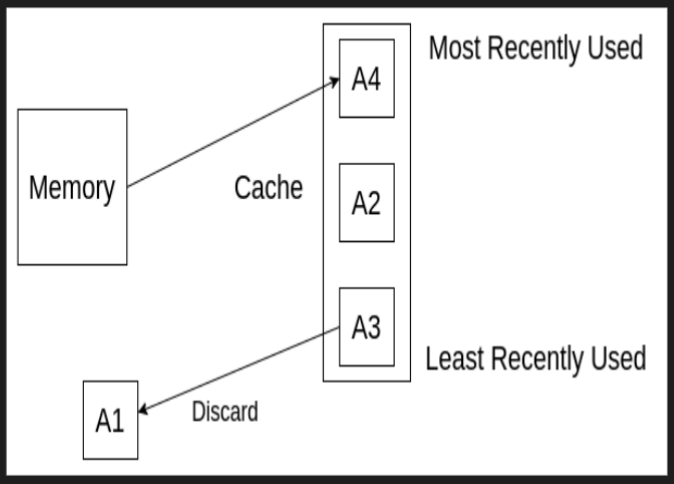
Persistence with ChromaDB&SQLite

- Why ChromaDB &SQLite?
  - Vector database
  - Local Persistence
  - Scalability



Size and Eviction from Cache

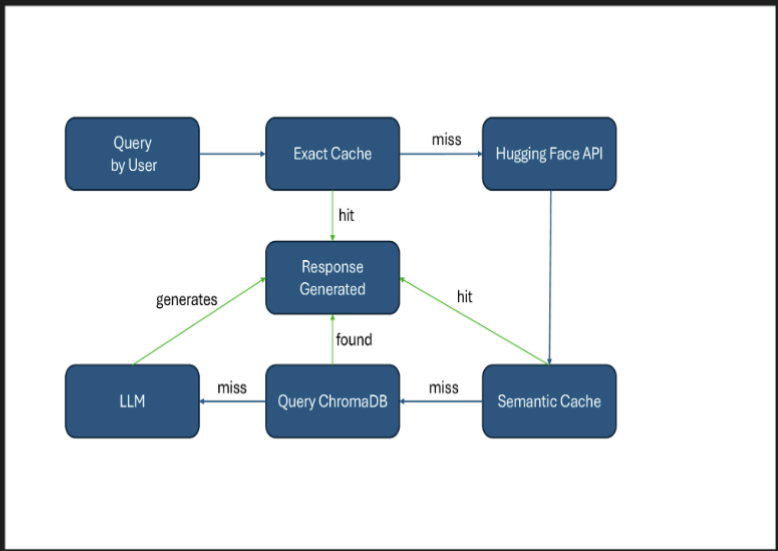
- LRU Strategy
  - manages the size of the caches
  - Prioritizing Recent Data
  - Balance memory and performance



Hugging Face API

- Why hugging face?
  - Pre-trained Language Models
  - Semantic Understanding

- API Integration Flow

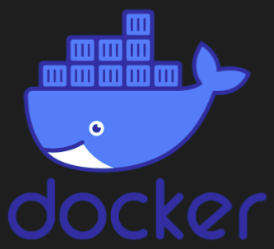


Challenges & Solutions

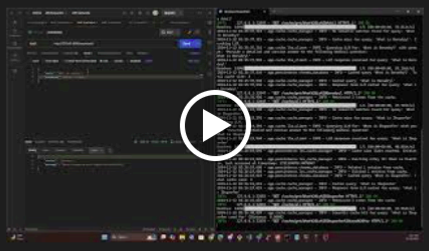
- Using both Git lab and a client-provided Github
  - Pushing to both repositories
- Chroma DB persistence integration
- PII masking
  - Remove specific medical information
    - Insurance policy numbers, dates of hospital visits
- No client LLM provided yet
  - Made mock queries and responses to test

Remaining Work & Timeline

- Connection to MedLib's frontend / access to their LLM
- Finish implementation of no-PII caching
- Additional logging and unit/integration testing
- Improvements to cache management/configuration via API
- Dockerize and deployment
- Feedback and additional features communicated by client



Demo



Problem Statement

- Problem: Specific focus on drug interaction queries
  - The complexity increases the computational burden on the LLM, especially when it needs to process large, unstructured medical datasets.
  - Lack of Efficiency Without Caching:
    - Without optimization, frequently asked drug-related queries or variations of the same query can repeatedly invoke the LLM.
    - This leads to redundant computation, increased latency, and higher operational costs.
  - Accuracy Concerns:
    - Drug interaction queries are highly sensitive. A minor misunderstanding or inaccurate response can have serious implications for patient safety.

Problem Statement

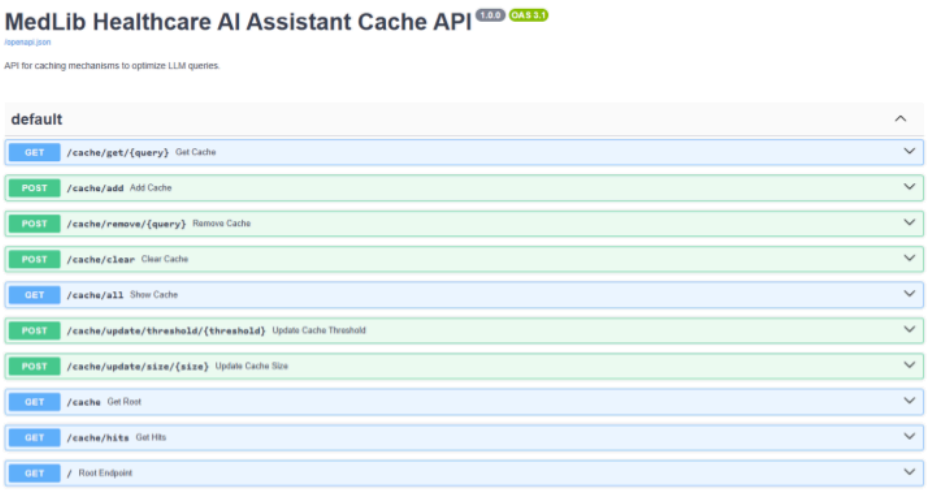
- Problem 1. Optimizing LLM performance
  - Caching Mechanism(Semantic & Exact cache)
    - store and retrieve responses for frequently queried terms
    - Developed a semantic cache to handle queries with similar meanings, reducing the reliance on repeated LLM processing
  - Similarity Threshold for Semantic Queries.
    - Allows fine-tuning between accuracy and efficiency
- Problem 2. Cost and Latency
  - Reduced API calls
    - Cached results provide near-instant responses, significantly improving query response times compared to LLM invocations
  - Processing & Embedding model
    - Integrated “SentenceTransformers” to compute embeddings locally, accelerating semantic query resolution and reducing external dependencies.

Objectives

Three-step process to cache previous queries to MedLib's LLM

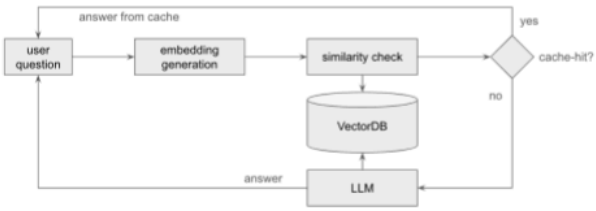
1. Store every query/response pair that is sent through the frontend
    a. Query: What are the side effects of sertraline?
    b. Response (from LLM): Side effects include insomnia, dizziness, etc...
2. Retrieve cached response when identical or similar query is asked
    a. Query 1: What happens if I take sertraline?
    b. Query 2: Are there side effects of sertraline?
    c. Response (from cache): Side effects include insomnia, dizziness, etc...
3. Return cached response if present, or signal to generate a new response through API

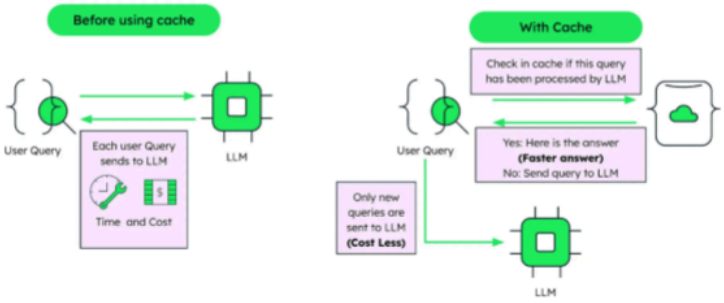
System Design - API Design (Jason)



System Design Overview

- Query received
- Check for exact or semantic cache match
- LLM call for non-cached queries





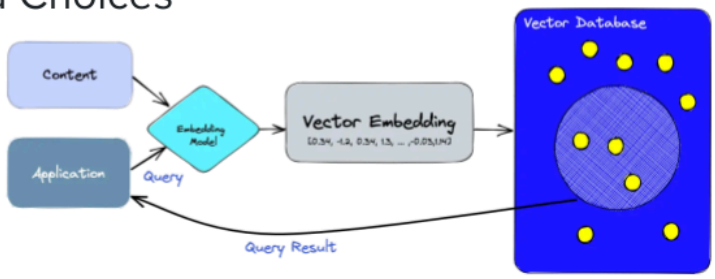
Design Challenges and Choices

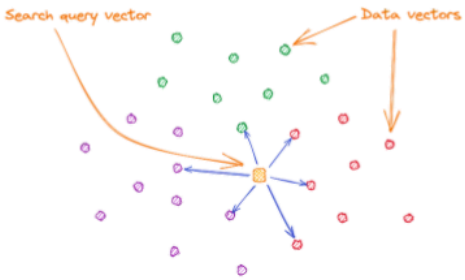
Challenges:

- Setting appropriate similarity thresholds
- Efficient cache eviction

Design Choices:

- SentenceTransformers for embeddings
- LRU for eviction policy
- ChromaDB for persistence





Chroma

Software Development Practices

Agile methodology with iterative development

Tools used:

- **Programming**: Python.
- **Frameworks**: FastAPI.
- **Embedding Models**: SentenceTransformers:

  all-MiniLM-L6-v2

- **Version Control**: Git/GitHub.
- **Containerization**: Docker.
- **Testing**: Unit and integration testing.













Accomplishments

Major Accomplishments:

- Implemented exact and semantic caching.
- Integrated cache with FastAPI endpoints.

Minor Accomplishments:

- Configurable similarity threshold.
- API for cache management.
- Implemented PII masking for privacy.

Highlight what team is proud of:

- Persistent storage with ChromaDB.
- Scalable design for future deployments.

Challenges Encountered

- Testing caching system without LLM
- Getting auth0 to function for the team
- Ensuring different types of PII get masked
  - i.e. names, addresses, phone numbers
  - Medical based information, such as insurance numbers
- Scheduling meetings between all 6 members and the client

Summary of Team Contributions

- Mohamed: Documentation and preparation, review on code.
- Jagger: Developed the semantic caching mechanism, integrated it with ChromaDB for persistent storage, implemented LLM API interaction, unifying exact and semantic caching, and unit testing.
- Owen: Worked on the semantic caching mechanism, implementing Least Recently Used eviction policy. Implemented caching modifications such as size and threshold, and statistics such as cache hit/misses.
- Jason: Set up project infrastructure, design and implement API system and routing, developed exact caching mechanism, and prepared docker and deployments
- Chanho**:** Set up unit tests for Exact Cache and Integration of API endpoints. Also, set up the github action CI. Implemented Auth0 for centralizing the authentication management and simplifying the security
- Steven: Fully implemented PII masking for caching responses, set up unit test to ensure masking works, assisted with exact cache setup.