Project Discussion & Design Decisions

- Ishwar Ambad

GitHub Link - https://github.com/ish-war/infraintel-medical-assistant

1. Problem Understanding

The project began with the requirement to build a **Retrieval-Augmented Generation (RAG)-based medical assistant**. The goal was to process **unstructured patient medical records**, summarize them, and enable efficient **semantic search + AI-assisted answering** over the data.

The challenge was twofold:

1. Designing a pipeline that ingests and processes documents from **Google Cloud Platform (GCP) Document AI output**.

2. Building a **scalable agent service** that retrieves relevant summaries and optionally synthesizes answers using an LLM.

---

2. Data Ingestion & Storage

- **Source:** Raw documents (medical test reports, patient summaries) were stored in a **GCP Storage Bucket**.

- **Access & Authentication:**

    - We created a **dedicated service account** (`my-docai-sa@document-ai-473404.iam.gserviceaccount.com`) with **Document AI & Storage permissions**.

    - A **key file (`.json`)** was generated and secured locally as `key.json`.

- The path was exported via the `GOOGLE_APPLICATION_CREDENTIALS` environment variable.

This design ensured **secure access** without embedding credentials directly in code.

---

3. Document Summarization (Task 1 → Task 2)

Design Decision:

Instead of storing bulky raw text, we chose to generate **summarized, structured outputs**. This improved **retrieval efficiency** and avoided LLM hallucination on irrelevant data.

- **Summarization pipeline:**
  - Extracted **key patient details** (Name, Diagnosis, Treatment, Doctor Notes).
  - Stored both the **structured summary JSON** and **textual summary**.
  - Each processed record was time-stamped and assigned a unique `record_id`.

Why this approach?

- Easier to query specific metadata.
- Reduced noise in RAG search space.
- Faster embeddings computation.

---

4. Embedding & Vector Store (FAISS)

Once summaries were available, we needed **semantic search**.

- Chosen approach: **FAISS (Facebook AI Similarity Search)**.
- **Design choice:** Store embeddings + metadata separately for flexibility.

Artifacts generated:

- `summary_index.index` → FAISS binary index
- `summary_texts.npy` → Encoded text embeddings
- `summary_metadata.json` → Patient metadata aligned with embeddings

Why FAISS?

- High-performance similarity search.
- Lightweight and easy to integrate with FastAPI.
- Local index file → portable, avoids external DB overhead.

---

5. RAG Agent (Task 3)

The **RAG Agent** was designed as the core user-facing service.

Components:

- **Retriever:** Searches FAISS for top-k relevant summaries.
- **LLM Layer (optional):** Synthesizes user-friendly answers using **Gemini 2.5 Flash** model.
- **Fallback Handling:** If Gemini is disabled/unavailable, raw retrieved summaries are returned.

Design Decision:

We added a `use_gemini` flag in the API to give flexibility:

- `true` → Use RAG + Gemini for natural answers.
- `false` → Return raw retrieved summaries only.

This made the system **robust to API failures** and **cost-flexible**.

---

6. API & UI Layer

Built using **FastAPI** for:

- REST endpoints (`/ask`, `/api`) → For programmatic integration.
- UI endpoints (`/`, `/ask-ui`) → For quick manual queries.

Design Considerations:

- **FastAPI** was chosen due to:
  - Easy async support.
  - Built-in OpenAPI/Swagger docs (`/docs`).
  - Lightweight deployment with Uvicorn.
- **UI** built with Jinja2 templates and Bootstrap for a professional yet simple look.

---

7. Deployment & Cloud Run

Dockerization

- Created a lightweight `Dockerfile` with Python, FAISS, FastAPI, and Uvicorn.
- Bundled vectorstore + service code.

Deployment

- Container pushed to **Artifact Registry**.
- Deployed on **Cloud Run** with flags:

- `--platform managed`
- `--allow-unauthenticated`
- `--timeout 600` (to avoid cold start failures).

Challenge:

- Cloud Run automatically sets `PORT` environment variable.
- Initially, setting `PORT=8080` manually caused deployment errors.
- **Fix:** Removed explicit `PORT` from `--set-env-vars`. FastAPI was updated to read `PORT` dynamically.

---

8. Key Thought Process & Design Principles

- **Security first:** Used service accounts & `.env` configs, avoided hardcoding secrets.
- **Efficiency:** Summarization reduced irrelevant context before RAG embedding.
- **Scalability:** FAISS chosen for local prototyping, but pipeline allows migrating to **Pinecone / Vertex AI Matching Engine** later.
- **Flexibility:** Toggle between *raw retrieval* vs *LLM synthesis*.
- **Resilience:** Designed fallbacks when APIs fail.

---

9. Future Improvements

- Integrate **Vertex AI Matching Engine** for large-scale search.
- Add **role-based authentication** in FastAPI.

- Improve **summarization templates** to capture additional medical context.
- Monitor query performance and optimize FAISS index refresh strategies.

10. Conclusion

This project demonstrates the **end-to-end** lifecycle of building a cloud-native, AI-powered RAG system:

- Automated ingestion of medical data from GCP buckets
- Summarization pipeline that transforms raw reports into structured outputs
- Embedding and FAISS-based retrieval for semantic search
- A FastAPI-powered RAG agent with optional LLM synthesis
- Successful containerization and deployment on GCP Cloud Run
- Improve **summarization templates** to capture additional medical context.
- Monitor query performance and optimize FAISS index refresh strategies.

The design choices prioritized **security, scalability, and robustness**, ensuring that the assistant can handle real-world medical data workflows. By combining **Document AI, FAISS, FastAPI, and Cloud Run**, we created a modular system that can be extended with advanced LLM models and production-grade search engines.

This work lays the foundation for **scalable medical knowledge assistants** that can empower both healthcare professionals and patients with **quick, reliable, and secure access to medical insights**.