**MedView Database Documentation**

Github Repo: <https://github.com/harrrisw/medviewdata_local>

**MedView Database setup (mdvn)**

This file implements the data layer of a Retrieval-Augmented Generation (RAG) system for MedView, a healthcare support platform. The pipeline enables semantic retrieval over medical device documents using vector embeddings and a vector database.

Goals & Features

- Embed unstructured text files into vector representations using OpenAI

- Split and store document chunks in AstraDB for fast similarity search

- Enable downstream components (e.g., Q&A chatbots) to retrieve contextually relevant content

Future Work

- Integrate environment variable handling (currently hardcoded)

- Support bulk file ingestion

- Add metadata tagging (device, language, section)

- Validate embedding integrity with unit tests

**User Documentation**

How to Use:

1. Clone the repository and install dependencies (langchain, openai, astrapy)

2. Place .txt files to embed in the project directory

3. Run mdvn.ipynb to vectorize the content and store in AstraDB

Required Setup:

Create a .env file or securely insert your credentials:

OPENAI\_API\_KEY=your\_key

ASTRA\_DB\_API\_ENDPOINT=your\_endpoint

ASTRA\_DB\_APPLICATION\_TOKEN=your\_token

**Technical Documentation**

Code Architecture:

Component Purpose

OpenAIEmbeddings -Converts text chunks into vectors

AstraDBVectorStore - Stores vectorized documents for retrieval

RecursiveCharacterTextSplitter -Splits large documents into overlapping chunks

load\_and\_vectorize\_text\_file -Reads .txt file, splits, embeds, stores in AstraDB

Key Libraries:

- langchain\_openai

- langchain\_astradb

- langchain\_text\_splitters

- os, dotenv (recommended)

Features & Functionality

- Embeds full-length documents

- Stores document chunks in AstraDB

- Enables plug-and-play vector retrieval for LLMs

- Modular load\_and\_vectorize\_text\_file() for reuse

Deployment Notes

- Clarity: Code and function naming are readable and intuitive

- Access: Code requires minor setup (API credentials + .txt file)

- Deployment: Intended to run in Jupyter, but can be migrated to Python script or service

**Web Scraper**

This file implements a recursive web scraper designed to collect and normalize internal hyperlinks from medical device manufacturer websites (e.g., ResMed, Philips). The collected links can be used to extract product pages, manuals, or support articles for later vectorization and retrieval.

Goals & Features

- Start from a given base URL and crawl internal links

- Normalize and deduplicate valid URLs

- Handle HTTP errors and retry failed connections

- Support recursion to a defined link depth

- Build a link database for downstream use in semantic embedding

Future Work

- Store scraped HTML for offline processing

- Filter and categorize pages by content type (manuals, FAQs, setup)

- Export to CSV/JSON for ingestion

- Add parallelism for large-scale scraping

**User Documentation**

How to Use:

1. Define a base\_url for the website you want to scrape.

2. Instantiate the AdvancedWebScraper class with desired depth and timeout.

3. Run get\_all\_links() to retrieve a list of unique internal links.

Example:

scraper = AdvancedWebScraper(base\_url="https://www.resmed.com")

all\_links = scraper.get\_all\_links()

**Technical Documentation**

Code Architecture:

Component Purpose

\_\_init\_\_() - Initializes base URL, depth, and request settings

get\_all\_links() - Starts scraping process and returns unique internal URLs

scrape\_site() - Recursively follows links up to max\_depth

extract\_links() - Parses anchor tags, normalizes URLs, filters internal links

requests + Retry - Adds robust retry logic for HTTP errors and timeouts

Error Handling:

- Built-in retry via HTTPAdapter and Retry

- Logs invalid URLs and connection errors

- Skips duplicates and malformed links

Features & Functionality

- Deep recursive scraping

- Internal link filtering

- Domain-restricted crawling

- Logging and retry logic for resilience

Deployment Notes

- Clarity: Clean class-based structure, easily reused

- Access: Requires only Python and requests + bs4

- Deployment: Can run as standalone scraper module or be integrated into a pipeline

**RAG Demo Data**

This file contains the owner’s manual for the FreeStyle Lite Blood Glucose Monitoring System.

It is used as input to the MedView retrieval-augmented generation (RAG) pipeline, which enables users to query product manuals

and receive precise answers via natural language.

Goals & Features

- Enable semantic search over the full FreeStyle Lite manual

- Provide both English and Spanish instructions

- Allow downstream components (e.g., chatbot) to answer user-specific queries about device setup, usage, and maintenance

- Improve accessibility and comprehension for end users

Future Work

- Add additional manuals (other glucose meters, CPAPs, etc.)

- Annotate and segment content by section titles for better retrieval accuracy

- Add embedded metadata (language, section, device) for multi-language filtering

- Integrate OCR pipelines for scanned documents

**User Documentation**

How to Use:

- This file is automatically split into chunks and embedded using OpenAI Embeddings.

- Chunks are stored in AstraDB and accessed via LangChain's retriever model.

- Users interact with a chatbot frontend that issues queries to this RAG system.

Example Use Case:

"How do I disinfect the FreeStyle Lite meter?"

→ Returns exact steps from the "Cleaning and Disinfection" section.

**Technical Documentation**

File Format:

- Plain text extracted from a medical device PDF manual

- Includes structured sections:

- Device setup, date/time config

- Blood glucose testing

- Troubleshooting, error codes

- Control solution usage

- Cleaning and disinfection

- Battery replacement

- Technical specs

Downstream Usage:

- Ingested into `mdvs` collection in AstraDB

- Embedded via `OpenAIEmbeddings`

- Queried via LangChain's `RetrievalQA` chain

Input Example for Vectorization:

RecursiveCharacterTextSplitter(chunk\_size=1000, chunk\_overlap=100)

Features & Functionality

- Rich procedural content

- Bilingual instructions (English + Spanish)

- Ideal for question-answering models

- Clean, structured format for semantic chunking

Deployment Notes

- Access: Stored as plain .txt for universal compatibility

- Security: Free of personal or sensitive information

- Deployment: Ready for vectorization and retrieval pipelines

**MongoDB NoSQL Pipeline**

This file uploads a collection of structured FAQs into MongoDB Atlas to support semantic search and retrieval for the MedView platform.

Each FAQ is associated with a specific CPAP machine model and is stored in a consistent, queryable format to support downstream NLP pipelines.

Goals & Features

- Store FAQs per medical device (e.g., AirSense 11, Prisma, Lumis)

- Organize data in a standard JSON schema

- Provide a database backend for the MedView chatbot and semantic search components

- Validate and verify MongoDB connection

Future Work

- Load data dynamically from external JSON/CSV sources

- Add update/delete capabilities via UI or API

- Expand schema to include tags, timestamps, or languages

- Improve error logging and retry handling

**User Documentation**

How to Use:

1. Ensure `pymongo` is installed.

2. Replace the hardcoded MongoDB URI with a secure method (e.g., environment variable).

3. Run the notebook to connect to MongoDB and insert structured FAQ documents.

Expected Output:

- Console confirmation of successful MongoDB connection.

- Documents inserted to the `mdvd.mdve` collection.

**Technical Documentation**

Code Architecture:

Component Purpose

MongoClient -Connects to MongoDB Atlas via connection string

insert\_data() -Uploads a predefined list of FAQ documents to MongoDB

Ping command -Verifies MongoDB connection

try/except - Catches and logs errors during DB operations

Data Schema Example:

{

"model": "AirSense 11",

"questions": [

{"question": "...", "answer": "..."},

...

]

}

Dependencies:

- pymongo

- ServerApi from pymongo (for MongoDB Atlas compatibility)

Features & Functionality

- Multiple device support (AirSense, Prisma, Lumis)

- Insert all Q&A pairs in a single call

- Structured document format for efficient search

- Reusable insertion function

Deployment Notes

- Replace credentials with `os.getenv()` or use `.env` file for security.

- Designed to be run once for batch data upload.

- Can be adapted into an API backend or data ingestion pipeline.

**Retrieval Model**

This file builds a Retrieval-Augmented Generation (RAG) pipeline for answering medical device questions. It integrates OpenAI embeddings, AstraDB as a vector store, and LangChain to retrieve relevant document chunks

and generate human-readable responses.

Goals & Features

- Enable question answering over embedded device manuals

- Connect a retriever to AstraDB for efficient semantic search

- Use OpenAI's GPT model to generate helpful answers with context

- Create a flexible retrieval chain using LangChain

Future Work

- Replace hardcoded credentials with environment variables

- Support multi-turn conversations and feedback loop

- Add evaluation metrics (e.g., retrieval precision, response quality)

- Deploy as an API for frontend integration

**User Documentation**

How to Use:

1. Set up environment variables for OPENAI\_API\_KEY, ASTRA\_DB\_API\_ENDPOINT, and ASTRA\_DB\_APPLICATION\_TOKEN.

2. Run the notebook to connect to AstraDB and build a retriever using pre-stored vectorized chunks.

3. Call `qa.run("Your question")` to receive an AI-generated answer.

**Technical Documentation**

Code Architecture:

Component Purpose

OpenAIEmbeddings - Converts input text into vectors

AstraDBVectorStore - Vector database backend to store/retrieve chunks

ChatOpenAI - OpenAI’s chat model (e.g., GPT-4)

RetrievalQA - LangChain wrapper to combine retriever + LLM

PromptTemplate - Customizes LLM prompts

Pipeline Flow:

- User query → vector retriever → top-k chunks → LLM with context → Answer

Example:

query = "How do I clean my ResMed CPAP machine?"

result = qa.run(query)

print(result)

Features & Functionality

- Uses LangChain's RetrievalQA chain

- Embeds and retrieves context using AstraDB

- Generates LLM-based answers using OpenAI

- Easily extendable to other models or data sources

Deployment Notes

- Move keys to .env file for security

- Can be wrapped into a Flask/FastAPI app

- Designed to scale with other devices or languages