ChatGPT

OmniQA-Chatbot: Functionality Summary and Enhancement Plan

Existing Project Functionality

1. Initialization and Configuration: The chatbot uses environment variables (see `templates/env_template.txt`) for API keys and settings (e.g. Pinecone index, Google Gemini API) [1]. Running `main.py` simply calls the `run_main` pipeline with a sample question [2]. The pipeline setup occurs in `pipeline/runner.py`, which first ensures the embedding model and vector store are ready via `initialize_resources()` [3]. This function calls `configure_moudles()` (defined in `pipeline/runner_service.py`) to initialize external services: it configures Google's Generative AI (Gemini) for embeddings and connects to Pinecone as the default vector store [4]. In summary, at startup the code loads the embedding model (`GeminiEmbeddings`) and a Pinecone vector index using the provided API keys.

2. Data Loading (Q&A Datasets): The system can load a QA dataset either from HuggingFace or from a Postgres database. In `pipeline/runner.py` the function `select_vectorstore` checks what data source is specified and prepares the vector store accordingly [5]. If a HuggingFace dataset name and column names are provided, it uses `load_dataset_from_hf()` to fetch the dataset and convert it into a DataFrame of "question" and "answer" columns [6]. Similarly, if a Postgres table and columns are specified, `load_data_from_postgres()` is called to retrieve data via the helper in `src/db_utils/postgres_utils.py` [7]. The DataFrame of Q&A pairs (regardless of source) is then converted into a list of Langchain `Document` objects: each document's content is formatted as `"Q: {question}\nA: {answer}"` [8]. These documents are inserted into Pinecone by `load_data_to_vectorstore()`, which uses LangChain's Pinecone integration to upsert vectors for each chunk [9]. If no new dataset is provided, the pipeline will attempt to load a previously saved Pinecone index via `load_existing_vectorstore()` [10]. *(Notably, document file support was not yet implemented – the README mentions "Document support coming soon"* [11] *– so currently the knowledge base is built from Q&A pairs rather than free-form documents.)*

3. Embeddings and Vector Store: Under the hood, OmniQA-Chatbot uses Google's PaLM (Gemini) for text embeddings and Pinecone as the vector database. The class `GeminiEmbeddings` (in `src/embeddings/gemini_embeddings.py`) calls the Google Generative AI API to get high-dimensional embeddings for documents and queries [12]. These embeddings are then stored/retrieved from Pinecone. The vector store logic lives in `src/db_utils/pc_utils.py`, where `upsert()` adds documents to the Pinecone index in batches [13] and `load_existing()` retrieves an existing index [14]. The environment variable `PINECONE_ENV` is used as the index name (e.g. `"qa-index"`) [15]. In short, when the system starts or new data is loaded, it creates/updates a Pinecone index with vector embeddings for each document (each document here being a Q&A pair). This vector store is later used to find relevant information for incoming questions via similarity search.

4. QA Chain (Self-Clarifying Question Answering): The core question-answering logic is implemented in `src/chains/qa_chains.py`, using a *Retrieval-Augmented Generation* approach. When a user asks a question, `run_main()` calls `run_pipeline()`, which invokes `run_chains()` (in `src/chains/__init__.py`, likely mapped to `QA_chain_with_websearch`) with the user question and the vector store [16] [17]. Inside `QA_chain_with_websearch`, the system first performs a **self-clarifying QA loop** via `self_clarifying_qa()` [18]. This loop works as follows: it queries the vector store for similar content and uses an LLM to attempt an answer, then possibly refines the question if needed [19] [20]. Specifically, in each iteration, `vectorstore.similarity_search(q, k=k_retrieval)` finds the top *k* relevant documents (Q&A pairs or text chunks) from the knowledge base [21]. These are concatenated as the `context` for the LLM. The **answer chain** LLM (using a model called `ChatGroq` with a "llama3-70b-8192" model) is run with the current question and retrieved context [22]. The prompt for this chain is defined in `src/templates/templates.py`: it instructs the LLM to answer *solely based on the provided context*, or respond with *"No suitable answer found in database."* if the context lacks the answer [23] [24]. The resulting answer is checked – if it signals that the database had no answer, the loop breaks [25]. Otherwise, the system uses a **clarification chain** (another LLM prompt) to decide if more information is needed: it feeds the original question and the draft answer into a prompt that either returns `"ENOUGH"` (if the answer is sufficient or cannot be improved) or generates a refined follow-up question [26]. If a follow-up question is produced (meaning more context might be needed), the loop repeats with this new question (essentially drilling down for more details) [20]. The loop continues until it either decides it has enough information or hits a maximum number of queries. All the Q&A pairs from these iterations are collected in a `history`. Finally, a **summary chain** LLM generates a concise final answer based on the gathered Q&A history [27]. This summary (which represents the answer derived from the knowledge base) is stored in `result['summary']` [28].

5. Web Search Integration: OmniQA can optionally augment answers with live web data if the local knowledge base is insufficient. If `web_mode=True` (as set in `main.py` [29]), after getting the summary from the knowledge base, the pipeline calls the web scraping tool. In `src/chains/qa_chains.py`, the `QA_chain_with_websearch` function checks `web_mode` and triggers a web search phase [18]. It uses `src.scraping.run()` (from `src/scraping/gemini_scraper.py`) to perform the search. The scraper code first uses a Generative AI model to decide if a web search is needed for the query (based on context) and to generate search keywords [30]. If needed, it runs a DuckDuckGo search and retrieves results [31] [32], possibly even fetching content from result URLs (the scraper has logic for static/dynamic pages) [33]. The retrieved web content or search snippets are then summarized by another LLM prompt into a `web_summary` [34]. Back in the QA chain, if the earlier knowledge-base answer was `"no suitable answer found..."`, the code will return the web summary directly as the answer [35]. Otherwise, it combines both sources: a **final chain** LLM takes the knowledge-base summary and the web summary and merges them into a final answer [36]. The final prompt (see `templates.py`) instructs the LLM to prioritize the corpus (internal data) answer but supplement with web info if relevant, and to cite the web content properly [37]. This combined answer is then returned as the chatbot's response. In summary, the system tries to answer from the internal Q&A data first and only uses web search as a fallback or enrichment when needed, giving a final answer that may include cited web information if the internal data was insufficient.

Proposed New Features and Tasks

Now that we understand the current system, we can plan the modifications. The goal is to **(a)** enable local document-based knowledge (using a local vector DB like FAISS or SQLite instead of Pinecone) and **(b)** create a configurable service (e.g. a FastAPI app) where users can load documents or datasets and interact with the chatbot via an endpoint. Below is a breakdown of tasks to achieve these enhancements:

A. Add Local Document Processing with FAISS/SQLite

1. **Implement Local Vector Store (FAISS or SQLite-based):** Introduce a local vector database to replace or complement Pinecone for document embeddings. For example, you can integrate **FAISS** via LangChain (e.g. using `FAISS.from_documents`) or use an embedded vector store like Chroma (which uses SQLite under the hood). Create a utility similar to the Pinecone utils – perhaps a new module `src/db_utils/faiss_utils.py` – with functions to save vectors and load them. This should mirror Pinecone usage: e.g. a function `upsert_faiss(docs, embeddings)` to build an index from documents, and `load_faiss_index()` to load an existing index from disk. (If using FAISS, you can store the index to a file for persistence; if using Chroma, you can specify a persist directory and just reuse the same directory to load existing data.)

2. **Parse and Embed Local Documents:** Create a pipeline to ingest files placed in a designated `docs/` folder (as you suggested for uploads). Write a function (e.g. `load_docs_from_folder(path)`) that reads all documents from `docs/` (supporting text files, PDFs, etc. as needed). For each file, you may need to split it into smaller chunks (especially if using large text files or PDFs – consider using a library or simple logic to split by paragraphs or a max token length). Convert each chunk into a LangChain `Document` object containing the chunk text and any metadata (e.g. file name, page number). Then, embed these documents using the existing embeddings model. Currently, `configure_moudles()` always sets up the Google Gemini embedding model [4]. You can continue to use that for embedding your documents (it will produce embeddings for each chunk). If an offline embedding model is preferred (to avoid external API calls), consider integrating a local model (e.g. SentenceTransformers via LangChain) as an optional step – but initially, using the existing model might be fine to keep scope manageable.

3. **Integrate FAISS/Local DB into the Pipeline:** Modify the selection logic in `pipeline/runner.py` so that it knows when to use the local vector store. For instance, you can add a parameter to `run_main()` like `use_local_docs=False` or a check for a `docs/` directory. If local docs are to be used, skip the Pinecone setup. In `initialize_resources()`, avoid initializing Pinecone when in local mode (no need for `Pinecone()` call or API key) – possibly refactor `configure_moudles` to make Pinecone optional. Then, in `select_vectorstore()`, add a branch: if local documents mode is on, call your new `load_docs_from_folder` to get the documents, then build a FAISS index with those. Essentially, instead of `load_data_to_vectorstore` which upserts to Pinecone [38], you will call `upsert_faiss(docs, embeddings)` to create a FAISS index. Ensure you handle persistence: e.g., save the FAISS index to disk so subsequent runs can use `load_faiss_index()` (similar to how `load_existing_vectorstore` loads Pinecone index [10]). You might use an environment flag or config setting to decide between Pinecone and FAISS – for example, if `PINECONE_API_KEY` is not set, default to local FAISS mode. Document this behavior clearly so it's easy to switch data backends.

4. **Adjust QA Chain for Document Content (if needed):** Since your current prompt format for context is Q&A pairs ("Q:... A:..."), consider how to handle plain document text. If you are using actual document passages as context, you might not format them with a Q/A prefix. It may be fine to feed them as raw text context – the answer chain prompt already says "Context: \n{context}\nUser question:" [39], which can accommodate any text. The LLM will attempt to find the answer in that context. So you might not need major changes to prompts – just ensure the context string is the gathered relevant snippets from documents. If you want, you can modify

the prompt's system message to reflect that the context might be from documents rather than a Q&A database (for clarity), but functionally it should work. Test the pipeline with a sample document to ensure that `vectorstore.similarity_search()` returns meaningful chunks and the LLM can answer from them. If the answers seem off, you might tweak chunk sizes or the prompt instructions slightly (for example, instructing the model to quote the document if needed or similar). Overall, the chain logic remains the same: retrieve top-k relevant chunks from the FAISS index and proceed with the self-clarifying loop and summary.

5. **Testing and Iteration:** Once integrated, test the local document pipeline. Place a few text files in the `docs/` folder and run the chatbot in local mode. Verify that the vector index is created (for FAISS, perhaps log the number of vectors or check the file size of the saved index) and that `similarity_search` returns expected results. Ask questions related to the document content and see if the answers are correct and only based on that content. This will likely surface any adjustments needed (e.g. if the model still says "No suitable answer..." when it should have found something, you might increase `k_retrieval` or adjust the prompt). Also test that if no docs are present or no relevant info is found, the system gracefully returns "No suitable answer found in database." as before.

B. Develop a Configurable Chatbot Service (FastAPI Endpoint)

1. **Set Up a FastAPI Application:** Create a FastAPI (or Flask, but FastAPI is suggested) app that will serve the chatbot. Make a new file, e.g. `service.py` or `app.py`, and initialize a FastAPI instance. You can structure it such that when the service starts, it runs any necessary setup (like loading the vector store). For example, on startup, you might call a function to prepare embeddings and load data: if using local docs, load the FAISS index or build it; if using a Postgres table or HuggingFace dataset as configured, load that. This could be done in a startup event handler or simply at the module level (so it executes on import). Make sure to handle configuration: perhaps use environment variables or a config file to specify which data source to use (e.g. `DATA_MODE=docs` vs `DATA_MODE=postgres`, etc., plus paths or dataset names). In a simple scenario, you might decide that if a `docs/` directory has files, use that; otherwise, fall back to a default dataset.

2. **Loading Data via an API (Optional):** For more flexibility, consider adding an endpoint to **configure or reload** the knowledge base at runtime. For example, a `POST /load-data` endpoint that accepts parameters like `"source": "docs"` or details for a Postgres connection or HuggingFace dataset name. Initially, if this is too complex, you can skip it and just rely on the presence of files or environment settings on startup. However, the feature goal is to let a person "load and configure the documents/postgres QA dataset" without code changes, which suggests an API or UI for that. As a first step, document the process: e.g. "Place your files in `docs/` and restart the service" or "set your Postgres table info in the .env file". Later, you can implement the API to reload data without restarting. If implementing now: define a request model for the config (could include: mode type, path or connection info, optional instructions text). In the endpoint function, call the appropriate loader (`load_docs_from_folder` or the existing `load_data_from_postgres`, etc.), rebuild the vector store (using the tasks from part A), and store it (perhaps in a global or FastAPI state) for use by the chat endpoint.

3. **Chat Endpoint Implementation:** Add an endpoint (e.g. `POST /chat` or `GET /chat`) that users can call with their questions. For example, a `POST /chat` could accept JSON like `{"question": "Your question here"}`. In the handler, extract the question and pass it to your QA pipeline. You can reuse the `run_main()` / `run_pipeline()` logic for answering, but

ensure it uses the already loaded vector store and embedding model rather than re-initializing each time. You might refactor `run_main` so that embeddings and vectorstore can be kept in memory and reused across calls (e.g. avoid calling `configure_moudles` on every request). One approach is to initialize these once at app startup and store them (perhaps as global variables or FastAPI `app.state` values). Then the chat endpoint handler can do something like: `answer = run_pipeline(question, vectorstore=my_vectorstore, embeddings=my_embeddings, web_mode=True)` – using the objects prepared earlier. If needed, you can create a lighter function than `run_main` to avoid re-loading data every call. The output from `run_pipeline` (or `QA_chain_with_websearch`) is likely a string (the final answer) or a dict with history and summary. Ensure you format the response nicely – probably just return the final answer text (and maybe the source citations if any, though in your current design the citations are baked into the text as per the prompt). You can wrap it in a JSON response like `{"answer": "<final answer text>"}`. Test this endpoint using a tool like curl or the interactive docs at `http://localhost:8000/docs` (which FastAPI provides) to ensure you get a response for sample questions.

4. **Include Instruction Customization:** Allow an optional **instruction prompt** to configure the chatbot's behavior or domain knowledge. This could be a text field a user can provide when setting up the service or calling the load-data endpoint. For instance, a user might specify instructions like *"Only answer based on the uploaded documents. Use a formal tone."* Internally, you can incorporate these instructions into the prompt templates. A straightforward way is to modify the system prompt in the **answer chain template**. In `src/templates/templates.py`, the answer template currently starts with a generic assistant role and context [39]. You could parametrize it to include an `{instructions}` variable. For example: *"You are a helpful assistant. {instructions}\n\nContext:\n{context}\nUser question:\n{question}\n..."*. If `instructions` is empty or not provided, it could default to an empty string or some neutral phrase. Implementing this may involve changing `make_templates()` to accept an instructions string, or you can post-process the template string before creating the `ChatPromptTemplate`. Another option (if using Chat models with system messages) is to prepend a system message containing the instructions before running the chain. Choose an approach that fits the architecture: if sticking with the current prompt templates, embedding the instructions as shown is easiest. Be sure to test that providing different instructions indeed alters the bot's answers or style as expected. Document for the user that this field is optional – if not given, the bot uses the default behavior ("helpful assistant" etc.).

5. **Testing the Service Locally:** Run the FastAPI app locally (e.g. with Uvicorn) and verify the end-to-end functionality. First, test with the default configuration (no special instructions, and using either some sample docs or a small dataset) – call the `/chat` endpoint and see that you get sensible answers. Next, place a new document in the `docs/` folder and restart or call the load-data endpoint (if implemented) to ingest it – then query the bot on information from that document to ensure it responds using the new data. Also test the behavior when web search is toggled: for example, ask a question unrelated to the docs to see if it falls back to web (if `web_mode` is enabled by default in the service). Finally, test the instructions parameter: configure a quirky instruction (like a role-play persona or a mandate to answer in a certain style) and confirm the responses follow it. This manual testing will validate that the two new features – local document QA and the API service – work together. It's a good idea to update the README or documentation to include usage of the new service (how to add files, how to query the endpoint, how to set instructions), ensuring that anyone (including your future self) can easily use the enhanced OmniQA-Chatbot.

Conclusion and Next Steps

By implementing the tasks above, you will transform OmniQA-Chatbot into a more versatile local QA system. The **core architecture** remains a Retrieval-Augmented Generator: now you can plug in local documents or databases as the knowledge source (using FAISS/SQLite for storage instead of Pinecone) and interact with the chatbot through a convenient web API. The outlined changes preserve existing capabilities (such as using Postgres or HuggingFace datasets) while adding the flexibility to work entirely offline with custom documents. Once these features are in place, further improvements could include refining the document chunking, adding an interface to upload files via the API, and possibly integrating a UI. For now, with the detailed understanding of the codebase (as summarized above) and the step-by-step tasks, you should be able to systematically implement the new functionalities and get the enhanced OmniQA-Chatbot up and running. Good luck with the project! [11] [19]

---

[1] env_template.txt
https://github.com/dark-horiznz/OmniQA-Chatbot/blob/d292652e3708963e03bc72d644b5f7782a475f49/templates/env_template.txt

[2] [29] main.py
https://github.com/dark-horiznz/OmniQA-Chatbot/blob/d292652e3708963e03bc72d644b5f7782a475f49/main.py

[3] [5] [10] [16] [17] [38] runner.py
https://github.com/dark-horiznz/OmniQA-Chatbot/blob/d292652e3708963e03bc72d644b5f7782a475f49/pipeline/runner.py

[4] [6] [7] [8] runner_service.py
https://github.com/dark-horiznz/OmniQA-Chatbot/blob/d292652e3708963e03bc72d644b5f7782a475f49/pipeline/runner_service.py

[9] [13] [14] [15] pc_utils.py
https://github.com/dark-horiznz/OmniQA-Chatbot/blob/d292652e3708963e03bc72d644b5f7782a475f49/src/db_utils/pc_utils.py

[11] README.md
https://github.com/dark-horiznz/OmniQA-Chatbot/blob/d292652e3708963e03bc72d644b5f7782a475f49/README.md

[12] embeddings.py
https://github.com/dark-horiznz/OmniQA-Chatbot/blob/d292652e3708963e03bc72d644b5f7782a475f49/experiments/scripts/embeddings.py

[18] [19] [20] [21] [22] [25] [27] [28] [35] [36] qa_chains.py
https://github.com/dark-horiznz/OmniQA-Chatbot/blob/d292652e3708963e03bc72d644b5f7782a475f49/src/chains/qa_chains.py

[23] [24] [26] [37] [39] templates.py
https://github.com/dark-horiznz/OmniQA-Chatbot/blob/d292652e3708963e03bc72d644b5f7782a475f49/src/templates/templates.py

[30] [31] [32] [33] [34] gemini_scraper.py
https://github.com/dark-horiznz/OmniQA-Chatbot/blob/d292652e3708963e03bc72d644b5f7782a475f49/src/scraping/gemini_scraper.py