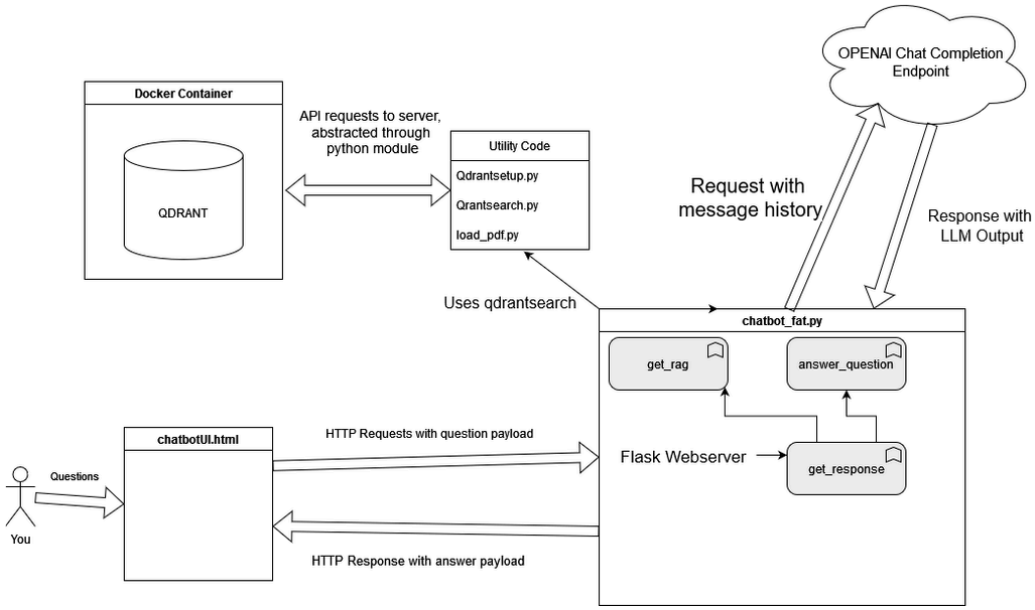
Design Overview



---

1. User Interface (chatbotUI.html)

- **Purpose**: This is the front-end interface where users interact with the chatbot.
- **Process**:
  - Users send their **questions** as input.
  - The interface sends HTTP requests (in the form of question payloads) to the Flask web server.
  - It receives HTTP responses containing the chatbot's answer.

---

2. Flask Web Server (chatbot_fat.py)

- **Purpose**: Acts as the backend, orchestrating communication between the user interface, the Qdrant vector database, and the OpenAI API.
- **Key Functions**:
  a. `get_response`:
    - Handles incoming requests from the chatbotUI and processes the question payload.
  b. `get_rag`:
    - Retrieves information relevant to the query using the Retrieval-Augmented Generation (RAG) approach, which involves querying the Qdrant database for context.
  c. `answer_question`:
    - Sends the combined question and retrieved context (message history) to the OpenAI Chat Completion Endpoint for generating a final answer.

---

3. OpenAI Chat Completion Endpoint

- **Purpose**: Provides the natural language generation capability.
- **Process**:
  - Receives the user question along with any relevant context retrieved from Qdrant.
  - Generates a detailed, coherent response using its large language model (LLM).
  - Sends the response back to the Flask server.

---

4. Qdrant (Docker Container)

- **Purpose**: A vector database used for storing and retrieving semantically meaningful information.
- **Process**:
  - The chatbot stores processed document data (e.g., PDFs) as vector embeddings in this database.
  - When a query comes in, Qdrant searches for the most relevant context based on semantic similarity to the question.

---

5. Utility Code (Python Modules)

- These Python scripts handle various backend tasks:
  - `Qdrantsetup.py`:
    - Sets up and manages the connection to the Qdrant vector database.
  - `Qdrantsearch.py`:
    - Handles search queries to retrieve relevant context from the database.
  - `load_pdf.py`:
    - Processes and loads PDFs into Qdrant by converting document content into vector embeddings.

Flow of Data:

1. **User Query**: The user enters a question in the chatbotUI.
2. **HTTP Request**: The question is sent as an HTTP request to the Flask server.
3. **Context Retrieval**:
   - Flask server uses `Qdrantsearch.py` to query Qdrant and retrieve relevant document context.
4. **LLM Response**:
   - The server sends the user query + retrieved context to the OpenAI endpoint.
   - OpenAI generates the final answer.
5. **HTTP Response**:
   - The Flask server sends the generated answer back to chatbotUI for display.