SATHI Project: System Architecture and Workflow

1. System Architecture Overview

The **SATHI Project** is built around a **Retrieval-Augmented Generation (RAG)** system. Its main goal is to let users ask questions through a chat interface and get accurate answers based on **private PDF documents**.

The system runs in two main phases:

Phase 1: Data Ingestion and Indexing

This is a **batch process** that prepares all the documents for efficient search and retrieval later on.

1. **Document Loading:**
   The system scans the /data directory and finds all PDF files.

2. **Text Extraction & Chunking:**
   Each PDF is processed using pdf_processor.py, which extracts the text.
   The text is then split into smaller, slightly overlapping chunks which helps capture semantic meaning more effectively during embedding.

3. **Vectorization (Embedding Generation):**
   Each chunk is converted into a **numerical vector** using the nomic-embed-text model (through ollama_client.py).
   These vectors represent the meaning of each chunk in high-dimensional space.

4. **Indexing in Qdrant:**
   The vectors (and their corresponding text chunks) are stored in a **Qdrant vector database** using qdrant_manager.py.
   Qdrant indexes the data so we can quickly find semantically similar text chunks later on.

Phase 2: Query Processing and Generation (RAG Pipeline)

This phase runs in **real time** whenever a user interacts with the chat interface.

1. **User Query:**
   The user enters a question in the Streamlit web app (app.py).

2. **FAQ Cache Check:**
   The system first checks if the query matches any preloaded FAQs (via faq_loader.py).
   If it finds a match, it instantly returns the cached answer — skipping the more complex retrieval process.

3. **Query Vectorization:**
   If no FAQ match is found, the system generates an **embedding vector** for the user's query using the same model as before (nomic-embed-text).

4. **Semantic Search (Retrieval):**
   That vector is compared with the document embeddings stored in Qdrant to find the top *k* most relevant text chunks.

5. **Prompt Augmentation:**
   The system combines these retrieved text chunks (the context) with the user's original question to create a single, well-structured prompt.

6. **Answer Generation:**
   This augmented prompt is sent to the local **LLM** (llama3.2:1b) via ollama_client.py. The model generates a response, streamed token-by-token to the Streamlit interface to simulate a "typing" effect.

---

2. Operational Workflows

Workflow 1: Data Ingestion

This process loads and indexes all the PDF data into Qdrant.

Trigger:
Run manually using:

python ingest.py

Steps:

1. ingest.py scans the ./data folder for all .pdf files.
2. Each file is processed with pdf_processor.extract_text_from_pdf() to extract text.
3. The text is chunked using pdf_processor.simple_text_splitter() based on the CHUNK_SIZE and CHUNK_OVERLAP settings.
4. A sample embedding is created via ollama_client.get_ollama_embedding() to determine the vector dimension.
5. A connection is established to Qdrant using qdrant_manager.get_qdrant_client().
6. The Qdrant collection (from config.py) is recreated, this clears all old data to ensure a clean index.
7. qdrant_manager.upsert_chunks() runs with a ThreadPoolExecutor to parallelize embedding generation and upsertion in batches.

Outcome:
The Qdrant database is now fully populated with embeddings of all document chunks and ready for semantic queries.

Workflow 2: Application Execution (Query Pipeline)

This describes what happens when the user interacts with the live app.

Trigger:
Run using:

streamlit run app.py

Steps:

1. The Streamlit app starts and initializes a Qdrant connection.
   If the target collection doesn't exist, it shows an error asking you to run ingest.py first.
2. FAQs are loaded from faqs.txt via faq_loader.load_faqs().
3. When the user submits a question (st.chat_input()):
   - **Path 1 (FAQ):** If the question matches an FAQ, the cached answer is returned immediately.
   - **Path 2 (RAG Pipeline):**
     1. Generate an embedding for the query (get_ollama_embedding()).
     2. Retrieve top *k* similar chunks from Qdrant (search_qdrant()).
     3. Construct an augmented prompt with the retrieved context.
     4. Stream a generated answer from llama3.2:1b using query_llama3().
     5. As tokens stream in, the app updates the chat display live.
4. The full conversation (query + answer) is stored in st.session_state.messages for the session.

---

3. Module and Component Overview

Group 1: Application & Documentation

- **app.py** – The main frontend app built with Streamlit.
  Handles chat state, user input, the RAG logic, and streaming responses.

- **README.md** – Developer documentation covering setup, dependencies, and commands for both ingestion and app execution.

Group 2: Data Ingestion

- **ingest.py** – Orchestrates the ETL pipeline:
  - **Extract:** Read and parse PDFs.
  - **Transform:** Generate embeddings.
  - **Load:** Store embeddings in Qdrant.

Group 3: Core Services (API Clients)

- **ollama_client.py** – Communicates with the local Ollama API.
  - get_ollama_embedding() – Calls /api/embeddings to generate embeddings.
  - query_llama3() – Calls /api/generate (with streaming) to get LLM responses.
- **qdrant_manager.py** – Handles all database operations with Qdrant.
  - get_qdrant_client() – Returns a configured client.
  - upsert_chunks() – Handles parallel embedding and batch uploads.
  - search_qdrant() – Performs semantic search and returns matching text chunks.

Group 4: Data Processing & Utilities

- **pdf_processor.py** – Handles PDF reading and text chunking.
  - extract_text_from_pdf() uses pypdf to read and combine text from pages.
  - simple_text_splitter() splits text into overlapping chunks for semantic continuity.
- **faq_loader.py** – Loads FAQs from a formatted text file (faqs.txt) into a dictionary for quick lookup.
- **config.py** – Stores key configurations such as service URLs, model names, and collection identifiers.
- **requirements.txt** – Lists project dependencies (streamlit, pypdf, qdrant-client, requests, etc.) to ensure consistent environments.