GCP Regulatory Chatbot - Comprehensive Code Documentation

Project Overview

This is a document-centric chatbot application designed for pharmaceutical regulatory compliance, specifically focusing on ICH GCP (Good Clinical Practice) guidelines. The system allows Quality Assurance and Regulatory Affairs professionals to interact conversationally with regulatory documents.

Project Architecture

- **Backend**: Python-based RAG (Retrieval-Augmented Generation) system
- **Frontend**: Streamlit web interface
- **AI Engine**: Local LLM via Ollama (Llama 3.2)
- **Vector Database**: ChromaDB for document embeddings
- **Document Processing**: LangChain for text processing and retrieval

---

1. Document Processor (`document_processor.py`)

Purpose

Core component responsible for document ingestion, processing, and vector storage management.

Key Functionalities

Class: DocumentProcessor

- **Initialization Parameters**:
  - `chunk_size=1000`: Size of text chunks for splitting documents
  - `chunk_overlap=200`: Overlap between consecutive chunks to maintain context
  - `embedding_model="sentence-transformers/all-MiniLM-L6-v2"`: Sentence transformer model for embeddings

Core Methods:

1. **`load_documents(file_paths)`**
   - Loads documents from specified file paths
   - Uses LangChain's TextLoader for UTF-8 encoded files
   - Adds metadata including source path, filename, and document type

    - Returns list of LangChain Document objects
    - Error handling for missing files
2. **`split_documents(documents)`**
    - Splits large documents into manageable chunks
    - Uses RecursiveCharacterTextSplitter with configurable separators
    - Adds chunk-specific metadata (chunk_id, chunk_index, total_chunks)
    - Maintains document coherence through overlapping chunks
3. **`create_vector_store(chunks, persist_directory)`**
    - Creates ChromaDB vector store from document chunks
    - Generates embeddings using HuggingFace sentence transformers
    - Persists database to disk for reuse
    - Stores metadata about chunks, models, and processing parameters
    - Saves configuration as JSON metadata file
4. **`load_vector_store(persist_directory)`**
    - Loads existing vector store from disk
    - Initializes ChromaDB with persisted embeddings
    - Loads metadata configuration
    - Returns boolean indicating success/failure
5. **`search_documents(query, k=5, filter_docs=None)`**
    - Performs similarity search on vector store
    - Supports filtering by specific document filenames
    - Returns top-k most relevant document chunks
    - Uses cosine similarity for relevance scoring
6. **`process_documents(file_paths, persist_directory)`**
    - Complete end-to-end document processing pipeline
    - Orchestrates loading, splitting, and vector store creation
    - Comprehensive error handling and logging

Technical Implementation Details

- **Text Splitting Strategy**: Uses multiple separators (paragraphs, sentences, punctuation) for intelligent chunking
- **Embedding Model**: Optimized for CPU usage with device='cpu' configuration
- **Vector Store**: ChromaDB provides efficient similarity search and filtering capabilities
- **Metadata Management**: Comprehensive tracking of document provenance and processing parameters

---

2. Chatbot Engine (`chatbot_engine.py`)

Purpose

Implements the conversational AI engine using RAG architecture with local LLM integration.

Key Functionalities

Class: GCPChatbot

- **LLM Integration**: Ollama-based local language model (Llama 3.2)
- **Memory Management**: ConversationBufferWindowMemory for context retention
- **RAG Implementation**: ConversationalRetrievalChain for document-aware responses

Core Methods:

1. `_initialize_llm()`
    - Configures Ollama LLM with specified model name
    - Sets temperature (0.1) for deterministic responses
    - Implements streaming callbacks for real-time output
    - Error handling for model availability
2. `_setup_prompt_template()`
    - Creates specialized prompt for GCP regulatory context
    - Structures prompts with context, chat history, and current question
    - Emphasizes accuracy, citation requirements, and professional tone
    - Instructs model to clearly state when information is unavailable
3. `load_documents(persist_directory)`
    - Loads pre-processed vector store
    - Initializes conversation memory with configurable window size
    - Creates ConversationalRetrievalChain linking LLM, retriever, and memory
    - Sets up document filtering and retrieval parameters
4. `set_selected_documents(document_names)`
    - Allows users to restrict search to specific documents
    - Creates filtered retriever for targeted document queries
    - Validates document availability before filtering
    - Updates retrieval chain with new filter constraints
5. `chat(user_input)`
    - Main conversation interface
    - Processes user queries through RAG pipeline
    - Extracts and formats source document references
    - Maintains conversation history with timestamps
    - Returns structured response with answer, sources, and metadata
6. `_get_chat_history_string()`
    - Formats recent conversation history for context
    - Limits to last 6 messages to manage prompt length
    - Provides human/assistant role identification

Advanced Features:

- **Document Filtering**: Custom retriever implementation for multi-document selection
- **Source Attribution**: Automatic extraction and formatting of document sources

- **Context Management**: Sliding window memory to maintain relevant conversation context
- **Error Handling**: Graceful degradation with informative error messages

Technical Implementation Details

- **RAG Architecture**: Combines document retrieval with generative AI for accurate, sourced responses
- **Local LLM**: Uses Ollama for privacy-compliant, offline operation
- **Memory Strategy**: Balance between context retention and computational efficiency
- **Prompt Engineering**: Specialized prompts for regulatory compliance use case

---

3. Streamlit Application (`streamlit_app.py`)

Purpose

Provides a user-friendly web interface for interacting with the GCP regulatory chatbot.

Key Functionalities

Class: StreamlitApp

- **Web Interface**: Streamlit-based responsive web application
- **Session Management**: Persistent state across user interactions
- **Document Management**: Interface for document selection and processing

Core Methods:

1. **`initialize_session_state()`**
   - Sets up Streamlit session variables
   - Manages chatbot instance, document loading status, chat history
   - Tracks selected documents and available document list
   - Ensures consistent state across page refreshes
2. **`setup_documents()`**
   - Checks for existing processed documents (ChromaDB)
   - Initializes chatbot if documents are available
   - Provides document processing interface for new setups
   - Validates file existence before processing
3. **`document_selection_sidebar()`**
   - Creates sidebar interface for document selection
   - Displays available documents with descriptions
   - Updates chatbot configuration when selection changes
   - Provides document metadata and information

4. **`chat_interface()`**
    - Main conversation interface
    - Displays chat history with proper formatting
    - Handles user input and response generation
    - Shows source documents for each response
    - Implements form-based input with submission handling
5. **`sample_questions()`**
    - Provides pre-defined questions for user guidance
    - Covers common GCP regulatory topics
    - Enables one-click question submission
    - Demonstrates chatbot capabilities

UI/UX Features:

- **Custom CSS**: Professional styling with color-coded message types
- **Responsive Design**: Mobile-friendly interface with proper spacing
- **Interactive Elements**: Expandable source citations and document information
- **Status Indicators**: Clear feedback on processing and loading states

Technical Implementation Details

- **State Management**: Efficient session state handling for web application persistence
- **Component Architecture**: Modular UI components for maintainability
- **Error Handling**: User-friendly error messages and fallback options
- **Performance**: Lazy loading and efficient re-rendering strategies

---

4. Requirements Management (`requirements.txt`)

Purpose

Defines all Python dependencies required for the application.

Key Dependencies

Core Framework:

- **streamlit>=1.28.0**: Web application framework
- **langchain>=0.0.350**: LLM application framework
- **langchain-community>=0.0.10**: Community extensions for LangChain

AI/ML Libraries:

- **ollama>=0.1.0**: Local LLM interface

- **sentence-transformers>=2.2.2**: Embedding model support
- **transformers>=4.35.0**: Hugging Face transformer models
- **torch>=2.0.0**: PyTorch for neural network operations

Vector Database:

- **chromadb>=0.4.18**: Vector database for embeddings
- **chroma-hnswlib>=0.7.3**: Hierarchical navigable small world graphs

Data Processing:

- **numpy>=1.24.0**: Numerical computing
- **pandas>=2.0.0**: Data manipulation and analysis
- **tiktoken>=0.5.0**: Token counting for text processing
- **unstructured>=0.10.30**: Document parsing utilities

Version Strategy

- Uses minimum version requirements (>=) for flexibility
- Ensures compatibility with latest security updates
- Balances stability with feature availability

---

5. Setup Script (`setup.py`)

Purpose

Comprehensive setup script for automated environment configuration and dependency installation.

Key Functionalities

System Validation Functions:

1. **`check_python_version()`**
   - Validates Python 3.8+ requirement
   - Exits gracefully if version incompatible
   - Provides clear version information
2. **`check_ollama_installation()`**
   - Verifies Ollama availability in system PATH
   - Tests Ollama functionality with version check
   - Returns installation status
3. **`install_ollama()`**
   - Provides platform-specific installation instructions

- Supports Linux, macOS, and Windows
- Guides users through manual installation process

Model Management:

1. **`download_llm_model(model_name)`**
   - Downloads specified LLM model via Ollama
   - Defaults to Llama 3.2 model
   - Provides download progress feedback
   - Handles network and storage errors

Environment Setup:

1. **`install_python_dependencies()`**
   - Installs all requirements from requirements.txt
   - Uses pip with proper error handling
   - Confirms successful installation
2. **`create_virtual_environment()`**
   - Creates isolated Python environment
   - Provides activation instructions
   - Platform-specific guidance

Document Processing:

1. **`process_documents()`**
   - Initializes DocumentProcessor
   - Processes Text_v1.txt and Text_v2.txt
   - Creates initial vector store
   - Handles processing errors gracefully

Setup Workflow

1. Python version validation
2. Required file verification
3. Ollama installation check/guidance
4. Python dependency installation
5. LLM model selection and download
6. Document processing
7. Success confirmation

---

6. Application Runner (`run_app.py`)

Purpose

Production-ready application launcher with comprehensive pre-flight checks.

Key Functionalities

System Health Checks:

1. **`check_ollama_running()`**
   - Verifies Ollama service status
   - Tests model list functionality
   - Timeout handling for unresponsive service
2. **`start_ollama()`**
   - Attempts to start Ollama service
   - Background process management
   - Service startup verification
3. **`check_dependencies()`**
   - Validates all required Python modules
   - Reports missing dependencies
   - Prevents runtime import errors
4. **`check_models()`**
   - Lists available Ollama models
   - Verifies model availability
   - Guides model download if needed

Application Launch:

1. **`run_streamlit_app()`**
   - Launches Streamlit with optimized configuration
   - Sets custom port and browser settings
   - Handles user interruption (Ctrl+C)
   - Provides clear startup feedback

Pre-flight Validation Sequence

1. Required file verification
2. Python dependency validation
3. Ollama service status check
4. Model availability verification
5. Application launch with monitoring

---

7. Document Content Analysis

Text_v1.txt (ICH E6(R2))

- **Document**: ICH E6(R2) - Good Clinical Practice Guidelines
- **Version**: E6(R2) (November 2016)
- **Content**: Comprehensive GCP guidelines including:
    - Principles of ICH GCP
    - IRB/IEC responsibilities and procedures
    - Investigator qualifications and duties
    - Sponsor responsibilities
    - Quality management systems
    - Essential documents and record keeping

Text_v2.txt (ICH E6(R3))

- **Document**: ICH E6(R3) - Good Clinical Practice Guidelines (Draft)
- **Version**: E6(R3) Draft (May 2023)
- **Content**: Updated GCP guidelines with:
    - Modernized principles for digital trials
    - Enhanced quality management approaches
    - Risk-based monitoring strategies
    - Data governance and computerized systems
    - Updated roles and responsibilities

Document Processing Strategy

- **Chunking**: 1000-character chunks with 200-character overlap
- **Indexing**: Semantic embeddings for context-aware retrieval
- **Metadata**: Source tracking, version identification, and chunk indexing
- **Search**: Similarity-based retrieval with document filtering

---

8. System Architecture Summary

Data Flow

1. **Document Ingestion**: Raw text files → Processed chunks → Vector embeddings
2. **Query Processing**: User question → Vector search → Context retrieval
3. **Response Generation**: Retrieved context + LLM → Formatted response
4. **Source Attribution**: Chunk metadata → Source citations

Technology Stack

- **Frontend**: Streamlit (Python web framework)
- **Backend**: Python with LangChain orchestration
- **AI Engine**: Local Ollama LLM (Llama 3.2)
- **Vector Store**: ChromaDB with HNSW indexing

- **Embeddings**: HuggingFace sentence-transformers

Key Design Principles

- **Privacy-First**: Local processing, no external API calls
- **Regulatory Compliance**: Audit trails and source attribution
- **Scalability**: Modular architecture for easy extension
- **User Experience**: Intuitive interface with clear feedback
- **Reliability**: Comprehensive error handling and validation

---

9. Deployment and Usage

Installation Process

1. Run `python setup.py` for initial setup
2. Execute `python run_app.py` to launch application
3. Access web interface at `http://localhost:8501`

User Workflow

1. **Document Selection**: Choose ICH E6(R2) and/or E6(R3)
2. **Query Submission**: Ask regulatory questions in natural language
3. **Response Review**: Receive answers with source citations
4. **Context Maintenance**: Continue conversation with memory retention

Operational Features

- **Document Filtering**: Multi-document selection capability
- **Source Transparency**: Direct citations to specific document sections
- **Conversation Memory**: Context-aware multi-turn dialogue
- **Export Capability**: Chat history export functionality

This comprehensive documentation provides a detailed understanding of each component in the GCP Regulatory Chatbot system, demonstrating a sophisticated RAG implementation for regulatory compliance use cases.