SmartDocs AI: Detailed Internship Project Task Breakdown

##

Milestone 1: Foundation & Core Setup (6 Tasks)

##

Task 1: Environment Setup & Project Initialization

Type: Backend Setup

Detailed Description:

Set up the complete development environment for the project. Install Python 3.8+, create a virtual environment using python -m venv venv, and activate it. Create a project folder structure with directories: backend/, frontend/, uploads/, data/, utils/, and config/. Install essential libraries: Streamlit, PyMuPDF, pdfplumber, openai, python-dotenv, chromadb, and langchain. Add them to requirements.txt. Initialize a Git repository and create a .gitignore file to exclude venv/, .env, uploads/, and __pycache__/. Create a simple test Streamlit app in frontend/app.py that displays "Hello World" to verify the setup. Document the setup process in README.md with installation and run instructions.

Deliverables:

Complete folder structure with all directories

requirements.txt with all dependencies

Git repository initialized with .gitignore

Test Streamlit app running successfully

README.md with setup instructions

Task 2: Basic PDF Text Extraction

Type: Backend Development

Detailed Description:

Develop a Python module backend/pdf_processor.py that extracts text from PDF files. Create a PDFProcessor class with methods: extract_text_pymupdf() for PyMuPDF extraction, extract_text_pdfplumber() as a fallback option, and get_pdf_metadata() for extracting page count, title, and author. The extraction function should read the PDF page by page and return a dictionary containing file name, total pages, and text content organized by page numbers. Implement error handling for corrupted files, password-protected PDFs, and empty documents using try-except blocks. Test with at least 3 different PDF types: simple text document, multi-column layout, and complex formatting. Create a test script test_extraction.py that processes these PDFs and prints extracted text with metadata to verify accuracy.

Deliverables:

backend/pdf_processor.py with PDFProcessor class

Functions for text extraction using both PyMuPDF and pdfplumber

Metadata extraction functionality

Error handling for common PDF issues

Test script demonstrating extraction from 3 sample PDFs

Task 3: Text Preprocessing & Cleaning

Type: Backend Development

Detailed Description:

Create a text preprocessing module backend/text_cleaner.py with a TextCleaner class. Implement functions: remove_extra_whitespace() to normalize spaces and line breaks, remove_special_characters() to clean unwanted symbols while keeping punctuation, remove_headers_footers() to eliminate page numbers and repetitive headers, and normalize_text() for case conversion and unicode handling. Each function should take text as input and return cleaned text. Chain these functions in a master method clean_text() that applies all cleaning steps in sequence. Handle edge cases like completely empty text, text with only special characters, and text with mixed encodings. Create a test file comparing before/after cleaning results with sample messy text containing excessive spaces, special characters, and formatting issues.

Deliverables:

backend/text_cleaner.py with TextCleaner class

Individual cleaning functions for different preprocessing tasks

Master clean_text() function that chains all cleaning steps

Test file showing before/after comparison with sample text

Documentation of what each cleaning function does

Task 4: Basic Streamlit UI - File Upload

Type: Frontend Development

Detailed Description:

Build the initial Streamlit interface in frontend/app.py with a professional layout. Create a sidebar using st.sidebar with a file uploader component (st.file_uploader) that accepts multiple PDF files with MIME type validation. Display application title, description, and instructions in the main area using st.title() and st.markdown(). Show uploaded file details (name, size in MB) in an organized table or list format. Add an "Process Documents" button that triggers file processing. Implement validation to check file types (only .pdf), file sizes (limit to 10MB per file), and display appropriate success messages using st.success() or error messages using st.error(). Add a file counter showing "X files uploaded". Include helpful instructions and tooltips guiding users on how to use the application. Style the UI with custom headers and spacing for better user experience.

Deliverables:

frontend/app.py with complete Streamlit UI layout

Sidebar with file uploader accepting multiple PDFs

File validation (type and size checks)

Display of uploaded file information

Process button with success/error messaging

User instructions and tooltips

Task 5: OpenAI API Integration & Testing

Type: Backend Development

Detailed Description:

Set up OpenAI API integration in the project. Create an OpenAI account at platform.openai.com and generate an API key from the API keys section. Create a .env file in the root directory and store the API key as OPENAI_API_KEY=your_key_here. Create a module backend/openai_helper.py with functions: load_api_key() to read the key from .env using python-dotenv, test_connection() to verify API connectivity by sending a simple prompt, and get_completion() as a wrapper for OpenAI API calls. Implement error handling for: invalid API keys, rate limit exceeded, network errors, and insufficient credits. Use the openai.ChatCompletion.create() method with model "gpt-3.5-turbo" for testing. Create a test script that sends a test prompt "Hello, are you working?" and prints the response. Document the API setup process including how to obtain and configure the API key in a separate API_SETUP.md file.

Deliverables:

.env file template (without actual key)

backend/openai_helper.py with API integration functions

API key loading and validation functions

Test script demonstrating successful API connection

API_SETUP.md with step-by-step API configuration guide

Error handling for common API issues

Task 6: Text Chunking Implementation

Type: Backend Development

Detailed Description:

Develop a text chunking module backend/text_chunker.py with a TextChunker class. Implement two chunking strategies: chunk_by_tokens() for fixed-size chunks (1000 tokens with 200 token overlap using tiktoken library), and chunk_by_sentences() for semantic chunking that preserves sentence boundaries. Create a function create_chunks() that takes cleaned text and returns a list of dictionaries, where each dictionary contains: chunk_id (unique identifier), text (chunk content), chunk_index (position in document), source_file (original PDF name), page_number (source page), and token_count. Implement overlap between chunks to maintain context continuity. Add metadata tracking for each chunk including character count and word count. Create utility functions: calculate_optimal_chunk_size() based on document length, and merge_small_chunks() to combine chunks that are too small. Test with documents of varying sizes (short 2-page PDF, medium 20-page PDF, long 100-page PDF) and verify chunks are properly segmented without breaking mid-sentence.

Deliverables:

backend/text_chunker.py with TextChunker class

Fixed-size and sentence-based chunking functions

Chunk metadata structure with all required fields

Overlap implementation to preserve context

Test script with multiple document sizes

Documentation explaining chunking strategies

##

Milestone 2: Vector Database & Retrieval System (6 Tasks)

##

Task 7: Vector Embeddings Generation

Type: Backend Development

Detailed Description:

Implement functionality to convert text chunks into vector embeddings in backend/embeddings.py. Create an EmbeddingGenerator class with methods: generate_embedding() to create a single embedding using OpenAI's "text-embedding-ada-002" model, and generate_batch_embeddings() to process multiple chunks efficiently with rate limiting (max 60 requests per minute). Implement add_delay() function to handle API rate limits by adding 1-second delays between batches. Create a function prepare_embedding_data() that takes a list of chunks and returns a list of dictionaries containing: embedding vector (1536-dimensional array), original text, chunk metadata, and timestamp. Add dimension verification to ensure all embeddings are 1536-dimensional. Implement caching mechanism to avoid regenerating embeddings for the same text. Handle API errors gracefully with retry logic (3 attempts with exponential backoff). Create a test script that generates embeddings for 10 sample chunks and verifies output dimensions, saves embeddings to a JSON file, and measures processing time.

Deliverables:

backend/embeddings.py with EmbeddingGenerator class

Single and batch embedding generation functions

Rate limiting and retry logic implementation

Embedding data structure with metadata

Dimension verification functionality

Test script with sample chunks and performance metrics

Task 8: Vector Database Setup - ChromaDB

Type: Backend Development

Detailed Description:

Set up ChromaDB as the vector database in backend/vector_db.py. Create a VectorDatabase class with initialization method that creates a persistent ChromaDB client stored in data/chroma_db/ directory. Implement create_collection() to set up a collection named "smartdocs_collection" with metadata fields. Create CRUD operations: add_documents() to insert embeddings with metadata (chunk text, file name, page number, chunk_id), get_collection_stats() to return total documents and collection info, delete_collection() to clear all data, and check_collection_exists() for validation. Define the collection schema with fields: ids (unique identifiers), embeddings (vector data), documents (original text), and metadatas (file info, page numbers). Implement update_document() for modifying existing entries. Add connection verification and error handling for database connection issues. Create a test script that: initializes the database, adds 5 sample embeddings, retrieves them, displays collection statistics, and verifies data persistence by restarting and checking if data still exists.

Deliverables:

backend/vector_db.py with VectorDatabase class

ChromaDB initialization with persistent storage

CRUD operations for managing embeddings

Collection schema definition with metadata fields

Connection verification and error handling

Test script demonstrating database operations and persistence

Task 9: Document Ingestion Pipeline

Type: Backend Integration

Detailed Description:

Build an end-to-end document ingestion pipeline in backend/ingestion_pipeline.py. Create a DocumentIngestion class that orchestrates the complete workflow: (1) accepts uploaded PDF files, (2) uses PDFProcessor to extract text, (3) uses TextCleaner to preprocess, (4) uses TextChunker to create chunks, (5) uses EmbeddingGenerator to create vectors, (6) uses VectorDatabase to store everything. Implement process_single_document() for one PDF and process_multiple_documents() for batch processing. Add a progress tracking system that returns percentage completion at each stage. Implement get_processing_status() that returns current stage, files processed, and estimated time remaining. Add comprehensive logging using Python's logging module to track each step with timestamps. Create rollback_on_error() function to handle failures gracefully and clean up partial data. Implement validation at each stage to ensure data quality. Create an integration test with 3 PDFs that runs the complete pipeline, displays progress updates, and verifies all data is correctly stored in ChromaDB with proper metadata linkage.

Deliverables:

backend/ingestion_pipeline.py with DocumentIngestion class

Complete workflow integrating all previous modules

Progress tracking and status reporting functionality

Logging system with detailed stage information

Error handling and rollback mechanisms

Integration test with multiple PDFs showing end-to-end flow

Task 10: Similarity Search Implementation

Type: Backend Development

Detailed Description:

Develop the similarity search functionality in backend/search_engine.py. Create a SearchEngine class with methods: search_similar_chunks() that takes a user query, converts it to an embedding using OpenAI, performs similarity search in ChromaDB, and retrieves top-k most relevant chunks (default k=5). Implement calculate_relevance_score() using cosine similarity to score each result. Add filter_by_threshold() to remove results below 0.7 similarity score. Create rerank_results() function to optionally reorder results based on additional factors like recency or source document. Implement search_with_filters() that allows filtering by specific documents or page ranges. Add get_context_window() to retrieve surrounding chunks for better context (1 chunk before and after the matching chunk). Create format_search_results() to structure output as a list of dictionaries containing:

chunk text, source file, page number, relevance score, and chunk position. Test with 10 diverse queries ranging from specific factual questions to broad conceptual queries, verify retrieval accuracy, and tune the similarity threshold based on results quality.

Deliverables:

backend/search_engine.py with SearchEngine class

Query-to-embedding conversion functionality

Similarity search with relevance scoring

Threshold filtering and reranking capabilities

Context window retrieval for better results

Test suite with diverse queries and accuracy evaluation

Task 11: Enhanced UI - Query Interface

Type: Frontend Development

Detailed Description:

Expand the Streamlit UI in frontend/app.py to include a comprehensive query interface. Create a main query section with st.text_input() for user questions and a "Search" button. Add a chat history display using st.container() that shows all previous Q&A pairs in chronological order with timestamps. Implement a session state using st.session_state to maintain conversation history across interactions. Display search results in expandable sections using st.expander() showing: answer text, source document names, page numbers, and relevance scores (displayed as progress bars or star ratings). Add a sidebar panel showing: number of documents loaded, total chunks in database, and search statistics (queries made, average response time). Implement "Clear History" and "New Session" buttons to reset conversation. Add loading spinners using st.spinner() during search operations with messages like "Searching documents...". Create tabs using st.tabs() to separate "Upload Documents", "Ask Questions", and "Search History" sections. Style the interface with custom CSS for better readability and professional appearance.

Deliverables:

Enhanced frontend/app.py with query interface

Text input and search button for questions

Chat history display with timestamps

Session state management for conversation persistence

Formatted result display with source attribution

Statistics panel and navigation tabs

Loading indicators and user feedback

Task 12: Context-Aware Response Generation

Type: Backend Development

Detailed Description:

Implement the core Q&A functionality in backend/qa_engine.py. Create a QAEngine class with method generate_answer() that: (1) receives user question and retrieved chunks, (2) constructs a detailed prompt for GPT including system instructions, (3) formats the context from retrieved chunks, (4) sends to OpenAI API, and (5) returns the generated answer. Design the prompt template to instruct GPT to: answer only from provided context, cite source documents with page numbers in the response, admit uncertainty if answer not found, and maintain a helpful, professional tone. Implement format_context() to structure retrieved chunks with source attribution in the prompt. Create add_conversation_history() to include previous Q&A pairs for context continuity (last 3 exchanges). Add extract_citations() to parse and highlight document references in the answer. Implement token management to ensure prompt doesn't exceed GPT model limits (8k tokens for gpt-3.5-turbo). Create validate_answer_quality() to check if response actually uses the provided context. Test with various question types: factual queries, comparative questions, multi-document synthesis, and unanswerable questions. Verify answers are accurate, well-cited, and contextually appropriate.

Deliverables:

backend/qa_engine.py with QAEngine class

Prompt engineering for context-aware responses

Context formatting and conversation history integration

Citation extraction and source attribution

Token management for API limits

Test cases covering different query types and validation

##

Milestone 3: Enhancement, Testing & Deployment (7 Tasks)

##

Task 13: Advanced UI Features & Styling

Type: Frontend Development

Detailed Description:

Enhance the Streamlit interface with advanced features and professional styling. Implement a PDF viewer in the sidebar using st.components.v1.iframe() or base64 encoding to display the selected PDF when a citation is clicked. Create a statistics dashboard at the top showing: total documents uploaded (with icon), total chunks stored, total queries made, and average response time using st.metric() with delta indicators. Add custom CSS using st.markdown() with unsafe_allow_html=True to style the app with: custom color scheme (primary and secondary colors), professional fonts (Google Fonts), rounded corners for containers, shadow effects for depth, and improved spacing. Implement a collapsible sidebar with expandable sections for different functionalities. Add tooltips and help icons using st.help() next to complex features. Create an "About" section explaining how the application works. Add keyboard shortcuts (Enter to submit query). Implement dark mode toggle option. Include animated loading states and smooth transitions. Add export functionality to download Q&A history as PDF or text file using download buttons. Test UI responsiveness and ensure it works well on different screen sizes.

Deliverables:

PDF viewer integrated in sidebar

Statistics dashboard with metrics

Custom CSS styling with professional theme

Tooltips and help documentation

Export functionality for Q&A history

Dark mode toggle option

Responsive design testing documentation

Task 14: Multi-Document Source Attribution

Type: Backend + Frontend Integration

Detailed Description:

Implement comprehensive source attribution system across backend and frontend. In backend/qa_engine.py, modify generate_answer() to include structured citations in the response using a special format like [Source: filename.pdf, Page: 5]. Create parse_citations() function to extract all citations from the generated answer and structure them as a list. In backend/search_engine.py, add get_source_distribution() to analyze which documents contributed to the answer and return percentage distribution. Update the frontend to display citations prominently: create colored badges/tags for each source document, show page numbers as clickable links, and display a "Sources Used" section below each answer listing all contributing documents. Implement highlight_citation() function that makes citations interactive - when clicked, it should show a preview of the relevant page or scroll to that section. Add a source confidence indicator showing how many chunks from each document were used. Create a visual chart using st.bar_chart() showing contribution percentage per document for multi-document queries. Test with queries requiring information from 2-3 different documents and verify all sources are properly tracked and displayed.

Deliverables:

Structured citation format in generated answers

Citation parsing and extraction functions

Source distribution analysis functionality

Interactive citation badges in UI

Clickable citations with content preview

Visual chart showing document contributions

Test cases with multi-document queries

Task 15: Error Handling & Validation

Type: Backend + Frontend Integration

Detailed Description:

Implement comprehensive error handling throughout the application. Create utils/error_handler.py with custom exception classes: PDFProcessingError, EmbeddingError, DatabaseError, APIError, and ValidationError. Wrap all file operations in try-except blocks with specific error messages. Add input validation: file size limits (max 10MB), file format validation (only PDFs), empty file detection, and corrupted file handling. Implement query validation: check for empty queries, maximum query length

(500 characters), and special character sanitization to prevent injection attacks. Add API error handling: rate limit detection with automatic retry after delay, insufficient credits warning, network timeout handling (30 seconds), and API key validation on startup. Create a centralized logging system using Python's logging module that writes to logs/app.log with levels: DEBUG, INFO, WARNING, ERROR. Display user-friendly error messages in Streamlit using st.error() instead of technical stack traces. Implement graceful degradation: if embeddings fail, offer text-only search; if database is down, show cached results. Add health check endpoint check_system_health() that verifies: database connection, API connectivity, and disk space. Test all error scenarios systematically and document expected behavior for each error type.

Deliverables:

utils/error_handler.py with custom exception classes

Input validation for files and queries

Comprehensive API error handling with retry logic

Centralized logging system

User-friendly error messages in UI

Graceful degradation strategies

System health check functionality

Error scenario testing documentation

Task 16: Performance Optimization

Type: Backend + Frontend Optimization

Detailed Description:

Optimize the application for better performance and user experience. Implement caching using @st.cache_data decorator for expensive operations: PDF text extraction, embedding generation, and database queries. Set appropriate TTL (time-to-live) for each cache type. Create backend/batch_processor.py for handling multiple file uploads efficiently - process files in parallel using Python's concurrent.futures.ThreadPoolExecutor with max 3 workers. Optimize database queries by adding indexing in ChromaDB and implementing pagination for large result sets (load 10 results at a time with "Load More" button). Implement lazy loading for UI elements - don't load full PDF content until user requests it. Add progress bars using st.progress() for long-running operations showing: file processing (0-40%), embedding generation (40-70%), database storage (70-100%). Optimize memory usage by processing large PDFs in chunks rather than loading entire file. Implement connection pooling for database to reduce connection overhead. Add query result caching - store last 10 query results in session state. Compress embeddings before storage to reduce database size. Test performance with: single large PDF (100+ pages), multiple small PDFs (10 files), and simultaneous operations, measuring response times and memory usage.

Deliverables:

Caching implementation with appropriate decorators

Batch processing with parallel execution

Database query optimization and pagination

Lazy loading for UI elements

Progress bars for all long operations

Memory optimization techniques

Query result caching system

Performance testing report with metrics

Task 17: Session Management & History

Type: Frontend + Backend Integration

Detailed Description:

Implement comprehensive session management and conversation history features. Create backend/session_manager.py with class SessionManager that handles: create_session() to initialize new session with unique ID, save_session_state() to persist conversation data, and load_session_state() to restore previous sessions. Use Streamlit's st.session_state to store: uploaded documents list, processed chunks count, query history (questions and answers with timestamps), and user preferences. Implement a "Chat History" tab in the UI displaying all Q&A pairs with: formatted timestamps, color-coded questions vs answers, and relevance scores. Add conversation export functionality with formats: plain text (.txt), markdown (.md), and PDF using reportlab library - include document sources and timestamps in export. Create "Session Management" sidebar with options: "Save Session" (store to local JSON file), "Load Session" (restore from JSON), "Clear Current Session" (reset all data), and "Session Statistics" (show query count, documents processed, time spent). Implement auto-save every 5 interactions. Add "Bookmark" feature to mark important Q&A pairs for later reference. Create session timeout mechanism (clear after 24 hours of inactivity). Test session persistence by: creating session, closing browser, reopening, and verifying data restoration.

Deliverables:

backend/session_manager.py with session handling

Session state management using Streamlit

Chat history display with formatting

Export functionality in multiple formats

Session save/load from JSON files

Auto-save and bookmark features

Session timeout mechanism

Persistence testing documentation

Task 18: Testing, Documentation & Deployment

Type: Testing + Documentation + Deployment

Detailed Description:

Conduct comprehensive testing and prepare project for deployment. Create tests/ directory with unit tests using pytest: test_pdf_processor.py (test extraction with various PDF types), test_text_cleaner.py (verify cleaning functions), test_chunker.py (validate chunking logic), test_embeddings.py (check embedding dimensions), test_vector_db.py (test CRUD operations), test_search_engine.py (verify search accuracy). Perform integration testing with tests/test_integration.py that runs complete pipeline from PDF upload to answer generation. Create test data folder with sample PDFs covering: simple text, tables, images, multi-column layouts, and large documents. Write comprehensive documentation: update README.md with project overview, features list, architecture diagram, installation steps, and usage instructions with screenshots. Create docs/USER_GUIDE.md with step-by-step tutorials for end users including: how to upload documents, ask questions, interpret results, and export history. Write docs/DEVELOPER_GUIDE.md explaining code structure, module descriptions, API references, and how to extend functionality. Document deployment process: create DEPLOYMENT.md for Streamlit Cloud deployment with environment variables setup and domain configuration. Prepare demo video (5-7 minutes) showing: UI walkthrough, document upload, query examples, and results interpretation. Create final presentation covering: project objectives, technical architecture, challenges faced, solutions implemented, demo, and future enhancements. Test deployment on Streamlit Cloud and verify all features work in production environment.

Deliverables:

Complete test suite with pytest (unit + integration tests)

Test data folder with sample PDFs

Comprehensive README.md with setup instructions

USER_GUIDE.md with tutorials and screenshots

DEVELOPER_GUIDE.md with code documentation

DEPLOYMENT.md with deployment instructions

Demo video (5-7 minutes)

Final presentation slides

Deployed application on Streamlit Cloud with public URL

Summary of Task Distribution

Milestone 1 (Foundation):

Backend Tasks: 4 (Tasks 1, 2, 3, 5, 6)

Frontend Tasks: 1 (Task 4)

Focus: Core infrastructure, PDF processing, API setup

###

Milestone 2 (Vector DB & Search):

Backend Tasks: 4 (Tasks 7, 8, 9, 10, 12)

Frontend Tasks: 1 (Task 11)

Focus: Embeddings, database, search, Q&A functionality

##

Milestone 3 (Enhancement & Deployment):

Backend Tasks: 2 (Tasks 15, 17)

Frontend Tasks: 2 (Tasks 13, 14)

Integration Tasks: 2 (Tasks 15, 16)

Testing/Deployment: 1 (Task 18)

Focus: Optimization, UI polish, testing, deployment

###

Total: 19 tasks (10 Backend, 4 Frontend, 2 Integration, 2 Backend+Frontend, 1 Testing/Deployment)