Complete RAG System Architecture for Wasserstoff Internship

Project Structure

```
chatbot_theme_identifier/
├── backend/
│   ├── app/
│   │   ├── api/
│   │   │   ├── __init__.py
│   │   │   ├── endpoints/
│   │   │   │   ├── __init__.py
│   │   │   │   ├── documents.py      # Document upload/management
│   │   │   │   ├── chat.py           # Query processing
│   │   │   │   └── themes.py         # Theme analysis
│   │   │   └── dependencies.py       # Database connections
│   │   ├── core/
│   │   │   ├── __init__.py
│   │   │   ├── config.py           # Environment variables
│   │   │   ├── database.py         # ChromaDB singleton
│   │   │   └── exceptions.py       # Custom exceptions
│   │   ├── models/
│   │   │   ├── __init__.py
│   │   │   ├── document.py         # Document schemas
│   │   │   ├── query.py            # Query/response schemas
│   │   │   └── theme.py            # Theme schemas
│   │   ├── services/
│   │   │   ├── __init__.py
│   │   │   ├── document_processor.py # PDF/OCR processing
│   │   │   ├── embedding_service.py  # Embeddings management
│   │   │   ├── retrieval_service.py  # Search & retrieval
│   │   │   ├── citation_manager.py   # Citation tracking
│   │   │   ├── theme_analyzer.py     # Theme identification
│   │   │   └── llm_service.py        # LLM interactions
│   │   ├── utils/
│   │   │   ├── __init__.py
│   │   │   ├── text_processing.py    # Text cleaning utilities
```

```
│   │   │   ├── chunking.py          # Document chunking
│   │   │   └── validation.py        # Data validation
│   │   └── main.py                  # FastAPI app
│   ├── data/
│   │   ├── uploads/                 # Document storage
│   │   ├── processed/               # Processed documents
│   │   └── embeddings/              # ChromaDB storage
│   ├── tests/
│   │   ├── test_api.py
│   │   ├── test_services.py
│   │   └── test_utils.py
│   ├── requirements.txt
│   ├── Dockerfile
│   └── docker-compose.yml
├── frontend/
│   ├── static/
│   │   ├── css/
│   │   ├── js/
│   │   └── uploads/
│   ├── templates/
│   │   ├── base.html
│   │   ├── upload.html
│   │   ├── chat.html
│   │   └── results.html
│   └── app.py                       # Streamlit frontend
├── docs/
│   ├── README.md
│   ├── ARCHITECTURE.md
│   ├── API_DOCS.md
│   └── DEPLOYMENT.md
├── scripts/
│   ├── setup_env.sh
```

```
│   ├── test_system.py
│   └── deploy.sh
└── demo/
    ├── demo_video.mp4
    └── sample_documents/
```

Technology Stack (All Free)

Core Technologies

- **Backend**: FastAPI (Python)
- **Frontend**: Streamlit
- **Database**: ChromaDB (vector database)
- **LLM**: Groq (free Llama models) + Ollama (local fallback)
- **Embeddings**: sentence-transformers (free)
- **OCR**: Tesseract + PaddleOCR
- **Clustering**: scikit-learn
- **Deployment**: Render (free tier)

Detailed Implementation

1. Document Processing Pipeline (`services/document_processor.py`)

python

```python
# Libraries needed:
# - PyMuPDF (fitz): PDF text extraction
# - Tesseract: OCR for scanned documents
# - PaddleOCR: Backup OCR
# - python-magic: File type detection
# - Pillow: Image processing

class DocumentProcessor:
    def __init__(self):
        self.ocr_engine = PaddleOCR(use_angle_cls=True, lang='en')
        self.tesseract_config = '--oem 3 --psm 6'

    def process_document(self, file_path: str, doc_id: str):
        """
        Step 1: File type detection
        Step 2: Text extraction (PDF/Text/Image)
        Step 3: OCR fallback for scanned content
        Step 4: Metadata extraction
        Step 5: Quality validation
        """

    def extract_pdf_text(self, pdf_path: str):
        """Use PyMuPDF for text extraction with fallback to OCR"""

    def extract_with_ocr(self, image_path: str):
        """PaddleOCR primary, Tesseract fallback"""

    def extract_metadata(self, file_path: str):
        """Extract title, author, creation date, page count"""
```

2. Text Chunking Strategy (`utils/chunking.py`)

python

```python
# Libraries:
# - spaCy: NLP processing
# - nltk: Sentence tokenization
# - tiktoken: Token counting

class SmartChunker:
    def __init__(self):
        self.nlp = spacy.load("en_core_web_sm")
        self.max_chunk_size = 512  # tokens
        self.overlap_size = 50

    def chunk_document(self, text: str, doc_metadata: dict):
        """
        Step 1: Section detection (headings, paragraphs)
        Step 2: Semantic boundary detection
        Step 3: Overlapping window creation
        Step 4: Chunk metadata assignment
        """

    def detect_sections(self, text: str):
        """Rule-based + NLP heading detection"""

    def create_semantic_chunks(self, sections: List[str]):
        """Respect paragraph boundaries, maintain context"""
```

3. Embedding Service (`services/embedding_service.py`)

python

```python
# Libraries:
# - sentence-transformers: Free embeddings
# - chromadb: Vector storage

class EmbeddingService:
    def __init__(self):
        self.model = SentenceTransformer('all-MiniLM-L6-v2')  # Free, fast
        self.db = ChromaDBSingleton.get_instance()

    def embed_document(self, doc_id: str, chunks: List[dict]):
        """
        Step 1: Generate embeddings for all chunks
        Step 2: Store in ChromaDB with metadata
        Step 3: Create document-level index
        Step 4: Validate storage
        """

    def embed_query(self, query: str):
        """Generate query embedding for retrieval"""
```

4. Retrieval Service (`services/retrieval_service.py`)

python

```python
# Libraries:
# - chromadb: Vector search
# - rank-bm25: Sparse retrieval
# - sentence-transformers: Cross-encoder reranking

class RetrievalService:
    def __init__(self):
        self.vector_db = ChromaDBSingleton.get_instance()
        self.reranker = CrossEncoder('cross-encoder/ms-marco-MiniLM-L-6-v2')

    def hybrid_search(self, query: str, k: int = 20):
        """
        Step 1: Vector similarity search (ChromaDB)
        Step 2: Keyword search (BM25)
        Step 3: Result fusion
        Step 4: Cross-encoder reranking
        Step 5: Relevance filtering
        """

    def get_diverse_results(self, results: List[dict]):
        """Ensure results from multiple documents"""
```

5. Theme Analysis (`services/theme_analyzer.py`)

python

```python
# Libraries:
# - scikit-learn: Clustering algorithms
# - umap-learn: Dimensionality reduction
# - hdbscan: Density-based clustering

class ThemeAnalyzer:
    def __init__(self):
        self.reducer = umap.UMAP(n_components=10, random_state=42)
        self.clusterer = hdbscan.HDBSCAN(min_cluster_size=3)
        self.llm_service = LLMService()

    def identify_themes(self, retrieved_chunks: List[dict]):
        """
        Step 1: Extract embeddings from chunks
        Step 2: Dimensionality reduction (UMAP)
        Step 3: Density-based clustering (HDBSCAN)
        Step 4: LLM theme naming for each cluster
        Step 5: Theme confidence scoring
        Step 6: Synthesized summary generation
        """

    def cluster_embeddings(self, embeddings: np.ndarray):
        """ML-based clustering, not LLM-dependent"""

    def name_themes_with_llm(self, clusters: dict):
        """Use LLM only for naming, not discovery"""
```

6. Citation Manager (`services/citation_manager.py`)

python

```python
# Libraries:
# - fuzzywuzzy: String matching
# - re: Regex for citation validation

class CitationManager:
    def __init__(self):
        self.document_store = {}  # Cache original documents

    def generate_citations(self, chunks: List[dict]):
        """
        Step 1: Extract source information from chunks
        Step 2: Verify text exists in original document
        Step 3: Calculate precise locations (page, paragraph)
        Step 4: Deduplicate same-source citations
        Step 5: Format citations consistently
        """

    def verify_citation_exists(self, chunk: dict):
        """Cross-reference with original document"""

    def deduplicate_citations(self, citations: List[dict]):
        """Merge citations from same document/page"""
```

7. LLM Service (`services/llm_service.py`)

python

```python
# Libraries:
# - groq: Free Llama API
# - ollama: Local LLM fallback
# - openai: API client format

class LLMService:
    def __init__(self):
        self.groq_client = Groq(api_key=os.getenv("GROQ_API_KEY"))
        self.local_model = "llama3.1:8b"  # Ollama fallback

    def answer_query(self, query: str, context_chunks: List[dict]):
        """
        Step 1: Format context with citations
        Step 2: Create structured prompt
        Step 3: Primary: Groq API call
        Step 4: Fallback: Local Ollama
        Step 5: Parse and validate response
        """

    def synthesize_themes(self, themes: dict, query: str):
        """Generate consolidated answer from themes"""

    def with_retry_and_fallback(self, prompt: str):
        """Robust API calls with fallback"""
```

8. API Endpoints (`api/endpoints/`)

Document Management (`documents.py`)

python

```python
# Libraries:
# - fastapi: Web framework
# - aiofiles: Async file handling
# - python-multipart: File uploads

@router.post("/upload")
async def upload_documents(files: List[UploadFile]):
    """
    Step 1: Validate file types and sizes
    Step 2: Save files to storage
    Step 3: Queue for processing
    Step 4: Return upload status
    """

@router.get("/documents")
async def list_documents():
    """Return all uploaded documents with metadata"""

@router.delete("/documents/{doc_id}")
async def delete_document(doc_id: str):
    """Remove document and embeddings"""
```

Chat Interface (`chat.py`)

python

```python
@router.post("/query")
async def process_query(query: QueryRequest):
    """
    Step 1: Validate query
    Step 2: Hybrid retrieval
    Step 3: Generate individual answers
    Step 4: Theme identification
    Step 5: Synthesized response
    Step 6: Return structured results
    """

@router.get("/query/{query_id}/results")
async def get_query_results(query_id: str):
    """Return formatted results for UI"""
```

9. Frontend (`frontend/app.py`)

python

```python
# Libraries:
# - streamlit: Web UI framework
# - requests: API calls
# - pandas: Data display

def main():
    st.title("Document Research & Theme Identification Chatbot")

    # Sidebar: Document management
    with st.sidebar:
        upload_documents()
        show_document_list()

    # Main area: Chat interface
    query = st.text_input("Ask a question about your documents:")

    if query:
        results = process_query_with_loading(query)
        display_results(results)

def display_results(results: dict):
    """
    Step 1: Individual document answers (table)
    Step 2: Theme-based synthesis (chat format)
    Step 3: Citation links and verification
    """
```

10. Configuration (`core/config.py`)

python

```python
# Libraries:
# - pydantic: Settings management
# - python-dotenv: Environment variables

class Settings(BaseSettings):
    # API Keys (free tiers)
    GROQ_API_KEY: str = ""

    # Database
    CHROMADB_PATH: str = "./data/embeddings"
    UPLOAD_PATH: str = "./data/uploads"

    # Processing
    MAX_FILE_SIZE: int = 50 * 1024 * 1024  # 50MB
    SUPPORTED_FORMATS: List[str] = [".pdf", ".txt", ".docx", ".png", ".jpg"]

    # Retrieval
    DEFAULT_K: int = 20
    RELEVANCE_THRESHOLD: float = 0.7

    class Config:
        env_file = ".env"
```

Deployment Strategy

Docker Setup (`Dockerfile`)

dockerfile

```dockerfile
FROM python:3.11-slim

# Install system dependencies
RUN apt-get update && apt-get install -y \
    tesseract-ocr \
    libtesseract-dev \
    libmagic1 \
    poppler-utils \
    && rm -rf /var/lib/apt/lists/*

WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt

COPY . .
EXPOSE 8000

CMD ["uvicorn", "backend.app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Free Deployment Options

1. **Render** (Recommended)
   - Free 750 hours/month
   - Automatic deployments from GitHub
   - Supports Docker
   - Built-in SSL
2. **Railway**
   - $5 free credit monthly

- Easy database integration
- GitHub integration

3. **Hugging Face Spaces**
   - Free for public projects
   - Great for Streamlit apps
   - GPU access available

Error Handling Strategy

Comprehensive Error Management

python

```python
# Custom exceptions for different failure modes
class DocumentProcessingError(Exception): pass
class EmbeddingError(Exception): pass
class RetrievalError(Exception): pass
class LLMError(Exception): pass

# Graceful degradation
def with_fallback(primary_func, fallback_func, error_types):
    """Generic fallback decorator"""

# Retry mechanisms for API calls
@retry(stop=stop_after_attempt(3), wait=wait_exponential(multiplier=1, min=4, max=10))
def call_groq_api(prompt: str): pass
```

Testing Strategy

Unit Tests (`tests/`)

```python
# Test each service independently
def test_document_processing():
    """Test PDF extraction, OCR, metadata"""

def test_chunking_strategy():
    """Test chunk quality and overlaps"""

def test_embedding_consistency():
    """Test embedding generation and storage"""

def test_citation_verification():
    """Test citation accuracy"""
```

Performance Optimizations

1. **Async Processing**: Use asyncio for I/O operations
2. **Caching**: Redis for frequent queries
3. **Batch Processing**: Process multiple documents together
4. **Connection Pooling**: Reuse database connections
5. **Lazy Loading**: Load embeddings on demand

Key Success Factors

1. **Reliability**: Every component has error handling and fallbacks
2. **Validation**: All citations verified against source documents
3. **Documentation**: Comprehensive setup and usage guides
4. **Testing**: Automated tests for all critical components

5. **Monitoring**: Logging and health checks throughout
6. **Scalability**: Designed to handle 75+ documents efficiently

This architecture addresses all the issues mentioned in your rejection feedback while staying within free-tier constraints.