Building a Free AI Document Search & Retrieval Assistant: Step-by-Step Guide

I'll outline how to build this solution using entirely free and open-source tools. This approach balances functionality with cost constraints while maintaining accuracy.

Prerequisites

- Basic Python programming knowledge
- Familiarity with command line interfaces
- Computer with at least 8GB RAM (16GB recommended)

Step 1: Set Up Your Development Environment (1-2 hours)

1. Install Python 3.10+ from python.org
2. Create a virtual environment:

bash

Copy

```
python -m venv docai-env

source docai-env/bin/activate  # On Windows: docai-env\Scripts\activate
```

3. Create a project folder structure:

bash

Copy

```
mkdir -p document-assistant/{data,models,src/{indexing,retrieval,summarization,api}}

cd document-assistant
```

Step 2: Document Processing Pipeline (1-2 days)

1. Install required libraries:

bash

Copy

```
pip install pypdf docx2txt python-pptx sentence-transformers nltk fastapi uvicorn
```

2. Create a document parser in src/indexing/parser.py:

python

Copy

```
import os

import pypdf

import docx2txt

from pptx import Presentation
```

```
def extract_text(file_path):
    """Extract text from various document formats"""
    _, ext = os.path.splitext(file_path)

    if ext.lower() == '.pdf':
        return extract_from_pdf(file_path)
    elif ext.lower() == '.docx':
        return docx2txt.process(file_path)
    elif ext.lower() == '.pptx':
        return extract_from_pptx(file_path)
    elif ext.lower() in ['.txt', '.md']:
        with open(file_path, 'r', encoding='utf-8') as f:
            return f.read()
    else:
        return None

def extract_from_pdf(file_path):
    text = ""
    with open(file_path, 'rb') as f:
        reader = pypdf.PdfReader(f)
        for page in reader.pages:
            text += page.extract_text() + "\n"
    return text

def extract_from_pptx(file_path):
    text = ""
    prs = Presentation(file_path)
    for slide in prs.slides:
        for shape in slide.shapes:
            if hasattr(shape, "text"):
                text += shape.text + "\n"
    return text
```

Step 3: Create Document Indexing System (1-2 days)

1. Create the vector embedding generator in src/indexing/embeddings.py:

python

Copy

```
from sentence_transformers import SentenceTransformer
import nltk
from nltk.tokenize import sent_tokenize
import numpy as np
import pickle
import os

# Download necessary NLTK data
nltk.download('punkt')

class DocumentIndexer:
    def __init__(self, model_name='all-MiniLM-L6-v2'):
        """Initialize with a lightweight sentence transformer model"""
        self.model = SentenceTransformer(model_name)
        self.documents = []
        self.embeddings = []
        self.chunks = []
        self.chunk_to_doc_map = []

    def add_document(self, doc_id, content, chunk_size=5):
        """Process and index a document by splitting into chunks"""
        sentences = sent_tokenize(content)

        # Create chunks of sentences
        for i in range(0, len(sentences), chunk_size):
            chunk = " ".join(sentences[i:i+chunk_size])
            self.chunks.append(chunk)
            self.chunk_to_doc_map.append(doc_id)

        # Store document
        self.documents.append({
            'id': doc_id,
            'content': content,
            'filename': os.path.basename(doc_id)
        })

    def generate_embeddings(self):
        """Generate embeddings for all document chunks"""
        self.embeddings = self.model.encode(self.chunks)

    def save_index(self, folder_path):
        """Save the index to disk"""
        os.makedirs(folder_path, exist_ok=True)

        with open(os.path.join(folder_path, 'documents.pkl'), 'wb') as f:
            pickle.dump(self.documents, f)

        with open(os.path.join(folder_path, 'chunks.pkl'), 'wb') as f:
            pickle.dump(self.chunks, f)

        with open(os.path.join(folder_path, 'chunk_to_doc_map.pkl'), 'wb') as f:
            pickle.dump(self.chunk_to_doc_map, f)

        with open(os.path.join(folder_path, 'embeddings.pkl'), 'wb') as f:
            pickle.dump(self.embeddings, f)
    @classmethod
    def load_index(cls, folder_path):
        """Load an existing index"""
        indexer = cls()

        with open(os.path.join(folder_path, 'documents.pkl'), 'rb') as f:
            indexer.documents = pickle.load(f)

        with open(os.path.join(folder_path, 'chunks.pkl'), 'rb') as f:
            indexer.chunks = pickle.load(f)

        with open(os.path.join(folder_path, 'chunk_to_doc_map.pkl'), 'rb') as f:
            indexer.chunk_to_doc_map = pickle.load(f)

        with open(os.path.join(folder_path, 'embeddings.pkl'), 'rb') as f:
            indexer.embeddings = pickle.load(f)

        return indexer
```

Step 4: Create the Indexing Script (1 day)

Create src/indexing/index_documents.py:

python

Copy

```
import os
import argparse
from parser import extract_text
from embeddings import DocumentIndexer

def index_documents(docs_folder, index_folder):
    indexer = DocumentIndexer()

    # Process each document in the folder
    for root, _, files in os.walk(docs_folder):
        for file in files:
            file_path = os.path.join(root, file)
            print(f"Processing: {file_path}")

            # Extract text from document
            text = extract_text(file_path)
            if text:
                indexer.add_document(file_path, text)
            else:
                print(f"Unsupported file format: {file_path}")

    # Generate embeddings for all documents
    print("Generating embeddings...")
    indexer.generate_embeddings()

    # Save the index
    print(f"Saving index to {index_folder}")
    indexer.save_index(index_folder)
    print(f"Indexed {len(indexer.documents)} documents with {len(indexer.chunks)} chunks")

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Index documents for search")
    parser.add_argument("--docs", required=True, help="Folder containing documents to index")
    parser.add_argument("--index", default="data/index", help="Folder to store the index")
    args = parser.parse_args()

    index_documents(args.docs, args.index)
```

Step 5: Build the Retrieval Engine (1-2 days)

Create src/retrieval/search.py:

python

Copy

```
import numpy as np
from sentence_transformers import SentenceTransformer

class DocumentRetriever:
    def __init__(self, indexer):
        """Initialize with a pre-built index"""
        self.indexer = indexer
        self.model = SentenceTransformer('all-MiniLM-L6-v2')

    def search(self, query, top_k=5):
        """Search for relevant document chunks"""
        # Generate embedding for the query
        query_embedding = self.model.encode(query)

        # Calculate cosine similarity between query and document chunks
        similarities = self._cosine_similarity(query_embedding, self.indexer.embeddings)

        # Get indices of top K most similar chunks
        top_indices = np.argsort(similarities)[-top_k:][::-1]

        # Get the corresponding document IDs and chunks
        results = []
        seen_docs = set()

        for idx in top_indices:
            doc_id = self.indexer.chunk_to_doc_map[idx]
            chunk = self.indexer.chunks[idx]
            similarity = similarities[idx]

            # Find the full document
            doc = next((d for d in self.indexer.documents if d['id'] == doc_id), None)

            if doc and doc_id not in seen_docs:
                seen_docs.add(doc_id)
                results.append({
                    'document': doc,
                    'relevance': float(similarity),
                    'preview': chunk
                })

        return results

    def _cosine_similarity(self, query_embedding, document_embeddings):
        """Calculate cosine similarity between query and document embeddings"""
        dot_product = np.dot(document_embeddings, query_embedding)
        query_norm = np.linalg.norm(query_embedding)
        doc_norm = np.linalg.norm(document_embeddings, axis=1)

        # Handle division by zero
        similarities = np.zeros(len(document_embeddings))
        valid_indices = np.where(doc_norm > 0)[0]
        similarities[valid_indices] = dot_product[valid_indices] / (query_norm * doc_norm[valid_indices])

        return similarities
```

Step 6: Create the Summarization Component (1 day)

Create src/summarization/summarizer.py:

python

Copy

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import sent_tokenize
from nltk.cluster.util import cosine_distance
import numpy as np
import networkx as nx

# Download necessary NLTK data
nltk.download('stopwords')
nltk.download('punkt')

class DocumentSummarizer:
    def __init__(self):
        self.stop_words = set(stopwords.words('english'))

    def summarize(self, text, num_sentences=5):
        """Generate an extractive summary of the text"""
        # Split text into sentences
        sentences = sent_tokenize(text)

        # Limit to reasonable number of sentences
        if len(sentences) <= num_sentences:
            return text

        # Calculate similarity matrix
        similarity_matrix = self._build_similarity_matrix(sentences)

        # Use PageRank to rank sentences
        scores = self._page_rank(similarity_matrix)

        # Select top sentences
        ranked_sentences = [(score, i, s) for i, (score, s) in
                          enumerate(zip(scores, sentences))]
        ranked_sentences.sort(reverse=True)

        # Get the indices of top sentences, then sort by position in document
        selected_indices = [i for _, i, _ in ranked_sentences[:num_sentences]]
        selected_indices.sort()

        # Create summary by joining selected sentences
        summary = ' '.join([sentences[i] for i in selected_indices])

        return summary

    def _build_similarity_matrix(self, sentences):
        """Build a similarity matrix based on sentence vectors"""
        # Initialize similarity matrix
        similarity_matrix = np.zeros((len(sentences), len(sentences)))

        for i in range(len(sentences)):
            for j in range(len(sentences)):
                if i == j:
                    continue
                similarity_matrix[i][j] = self._sentence_similarity(
                    sentences[i], sentences[j])

        return similarity_matrix

    def _sentence_similarity(self, sent1, sent2):
        """Calculate similarity between two sentences"""
        # Convert sentences to word vectors, ignoring stop words
        words1 = [word.lower() for word in nltk.word_tokenize(sent1)
                 if word.lower() not in self.stop_words]
        words2 = [word.lower() for word in nltk.word_tokenize(sent2)
```

```
                if word.lower() not in self.stop_words]

        # Create a set of all words
        all_words = list(set(words1 + words2))

        # Create word vectors
        vec1 = [1 if word in words1 else 0 for word in all_words]
        vec2 = [1 if word in words2 else 0 for word in all_words]

        # Calculate cosine similarity
        if sum(vec1) == 0 or sum(vec2) == 0:
            return 0
        return 1 - cosine_distance(vec1, vec2)

    def _page_rank(self, similarity_matrix, damping=0.85, max_iter=100, tol=1e-5):
        """Apply PageRank algorithm to rank sentences"""
        nx_graph = nx.from_numpy_array(similarity_matrix)
        scores = nx.pagerank(nx_graph, alpha=damping, max_iter=max_iter, tol=tol)
        return [scores.get(i, 0) for i in range(len(similarity_matrix))]
```

Step 7: Create the API (1 day)

Create src/api/main.py:

python

Copy

```
import os
from fastapi import FastAPI, Query, HTTPException
from pydantic import BaseModel
from typing import List, Optional
import sys

# Add parent directory to path
sys.path.append('..')
```

```
from indexing.embeddings import DocumentIndexer
from retrieval.search import DocumentRetriever
from summarization.summarizer import DocumentSummarizer

app = FastAPI(title="Document Search & Retrieval API")

# Initialize components
index_folder = os.environ.get("INDEX_FOLDER", "../../data/index")
indexer = None
retriever = None
summarizer = DocumentSummarizer()

@app.on_event("startup")
async def startup_event():
    global indexer, retriever
    print(f"Loading index from {index_folder}")
    try:
        indexer = DocumentIndexer.load_index(index_folder)
        retriever = DocumentRetriever(indexer)
        print(f"Loaded {len(indexer.documents)} documents with {len(indexer.chunks)} chunks")
    except Exception as e:
        print(f"Failed to load index: {e}")
        # Create empty index if not exists
        indexer = DocumentIndexer()
        retriever = DocumentRetriever(indexer)

class SearchQuery(BaseModel):
    query: str
    max_results: int = 5
    summarize: bool = False
```

```python
class SearchResult(BaseModel):
    document_name: str
    relevance: float
    preview: str
    summary: Optional[str] = None


@app.post("/search", response_model=List[SearchResult])
async def search(query: SearchQuery):
    if not retriever or len(retriever.indexer.documents) == 0:
        raise HTTPException(status_code=404, detail="No document index found. Please add documents first.")

    results = retriever.search(query.query, top_k=query.max_results)

    response = []
    for result in results:
        doc = result['document']
        item = SearchResult(
            document_name=doc['filename'],
            relevance=result['relevance'],
            preview=result['preview']
        )

        if query.summarize:
            item.summary = summarizer.summarize(doc['content'])

        response.append(item)

    return response
```

```
@app.get("/documents", response_model=List[str])
async def list_documents():
    if not indexer:
        raise HTTPException(status_code=404, detail="No document index found")

    return [doc['filename'] for doc in indexer.documents]

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

Step 8: Create a Simple Web UI (1-2 days)

Create src/api/static/index.html:

html

Copy

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document Search Assistant</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            max-width: 800px;
            margin: 0 auto;
            padding: 20px;
        }
        .search-container {
            margin-bottom: 30px;
        }
        input[type="text"] {
```

```
    width: 70%;
    padding: 10px;
    font-size: 16px;
}
button {
    padding: 10px 15px;
    background-color: #4285f4;
    color: white;
    border: none;
    font-size: 16px;
    cursor: pointer;
}
.options {
    margin: 10px 0;
}
.result {
    margin-bottom: 20px;
    padding: 15px;
    border: 1px solid #ddd;
    border-radius: 5px;
}
.result h3 {
    margin-top: 0;
}
.relevance {
    color: #4285f4;
    font-weight: bold;
}
.summary {
    background-color: #f9f9f9;
    padding: 10px;
```

```
            margin-top: 10px;
            border-left: 3px solid #4285f4;
        }
    </style>
</head>
<body>
    <h1>AI Document Search Assistant</h1>

    <div class="search-container">
        <input type="text" id="search-input" placeholder="Search documents...">
        <button id="search-button">Search</button>

        <div class="options">
            <label>
                <input type="checkbox" id="summarize-checkbox"> Generate summaries
            </label>
            <label>
                <input type="number" id="max-results" min="1" max="10" value="5"> Max results
            </label>
        </div>
    </div>

    <div id="results-container"></div>

    <script>
        document.addEventListener('DOMContentLoaded', function() {
            const searchInput = document.getElementById('search-input');
            const searchButton = document.getElementById('search-button');
            const summarizeCheckbox = document.getElementById('summarize-checkbox');
            const maxResults = document.getElementById('max-results');
            const resultsContainer = document.getElementById('results-container');
```

```
searchButton.addEventListener('click', performSearch);
searchInput.addEventListener('keypress', function(e) {
    if (e.key === 'Enter') {
        performSearch();
    }
});

function performSearch() {
    const query = searchInput.value.trim();
    if (!query) return;

    resultsContainer.innerHTML = '<p>Searching...</p>';

    fetch('/search', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify({
            query: query,
            max_results: parseInt(maxResults.value),
            summarize: summarizeCheckbox.checked
        })
    })
    .then(response => response.json())
    .then(data => {
        displayResults(data);
    })
    .catch(error => {
        resultsContainer.innerHTML = `<p>Error: ${error.message}</p>`;
    });
}

function displayResults(results) {
    if (results.length === 0) {
        resultsContainer.innerHTML = '<p>No results found.</p>';
        return;
    }

    resultsContainer.innerHTML = '';

    results.forEach(result => {
        const resultElement = document.createElement('div');
        resultElement.className = 'result';

        const relevance = Math.round(result.relevance * 100);

        resultElement.innerHTML = `
            <h3>${result.document_name}</h3>
            <p class="relevance">Relevance: ${relevance}%</p>
            <p><strong>Preview:</strong> ${result.preview}</p>
        `;

        if (result.summary) {
            resultElement.innerHTML += `
                <div class="summary">
                    <strong>Summary:</strong>
                    <p>${result.summary}</p>
                </div>
            `;
        }
```

```
                    resultsContainer.appendChild(resultElement);
                });
            }
        });
    </script>
</body>
</html>
```

Update src/api/main.py to serve the static files:

python

Copy

```
from fastapi.staticfiles import StaticFiles
from fastapi.responses import FileResponse

# Add at the start of your FastAPI app
app.mount("/static", StaticFiles(directory="static"), name="static")

@app.get("/")
async def read_index():
    return FileResponse("static/index.html")
```

Step 9: Create a Simple Document Upload Feature (1 day)

Update src/api/main.py to add document upload functionality:

python

Copy

```
import shutil
from fastapi import UploadFile, File
import tempfile
import sys

# Add to imports
from indexing.parser import extract_text
```

```
@app.post("/upload")

async def upload_document(file: UploadFile = File(...)):

    # Save file to temporary location

    temp_dir = tempfile.mkdtemp()

    temp_file_path = os.path.join(temp_dir, file.filename)



    try:

        with open(temp_file_path, "wb") as buffer:

            shutil.copyfileobj(file.file, buffer)



        # Extract text from document

        text = extract_text(temp_file_path)

        if not text:

            return {"status": "error", "message": "Unsupported file format"}



        # Add document to index

        indexer.add_document(temp_file_path, text)



        # Update embeddings

        indexer.generate_embeddings()



        # Save updated index

        indexer.save_index(index_folder)



        return {"status": "success", "message": f"Document '{file.filename}' added to index"}



    except Exception as e:

        return {"status": "error", "message": str(e)}



    finally:
        # Clean up temporary files
        shutil.rmtree(temp_dir)
```

Add upload form to src/api/static/index.html:

html

Copy

```
<!-- Add this after the search container -->
<div class="upload-container">
    <h2>Upload Document</h2>
    <input type="file" id="file-input">
    <button id="upload-button">Upload</button>
    <p id="upload-status"></p>
</div>

<!-- Add this to the JavaScript section -->
const fileInput = document.getElementById('file-input');
const uploadButton = document.getElementById('upload-button');
const uploadStatus = document.getElementById('upload-status');

uploadButton.addEventListener('click', uploadDocument);

function uploadDocument() {
    if (!fileInput.files[0]) {
        uploadStatus.textContent = 'Please select a file';
        return;
    }

    const formData = new FormData();
    formData.append('file', fileInput.files[0]);

    uploadStatus.textContent = 'Uploading...';
    fetch('/upload', {
        method: 'POST',
        body: formData
    })
    .then(response => response.json())
    .then(data => {
        uploadStatus.textContent = data.message;
    })
    .catch(error => {
        uploadStatus.textContent = `Error: ${error.message}`;
    });
}
```

Step 10: Package Everything Together (1 day)

Create a requirements.txt file:

Copy

```
fastapi==0.103.1
uvicorn==0.23.2
pypdf==3.16.2
docx2txt==0.8
python-pptx==0.6.21
sentence-transformers==2.2.2
nltk==3.8.1
networkx==3.1
python-multipart==0.0.6
```

Create a run.sh script:

bash

Copy

```
#!/bin/bash
# Activate virtual environment
source docai-env/bin/activate
```

```
# Set up environment variables
export INDEX_FOLDER="data/index"

# Create index directory if not exists
mkdir -p data/index

# Start the API server
cd src/api
uvicorn main:app --host 0.0.0.0 --port 8000 --reload
```

Step 11: Create Documentation (Half day)

Create a README.md file with usage instructions:

markdown

Copy

```
# AI Document Search & Retrieval Assistant

A free, open-source solution for intelligent document search, summarization, and retrieval.

## Features

- Natural language search queries
- Document summarization
- Support for PDF, Word, PowerPoint, and text files
- Web-based user interface
- API for integration with other applications

## Installation

1. Clone this repository
2. Install Python 3.10 or higher
3. Set up a virtual environment:
python -m venv docai-env source docai-env/bin/activate # On Windows: docai-env\Scripts\activate
```

Copy

4. Install dependencies:

```
pip install -r requirements.txt
```

Copy

## Usage

1. Start the application:

```
./run.sh
```

Copy

2. Open your browser to http://localhost:8000

3. Upload documents using the web interface

4. Search using natural language queries

## API Documentation

The API documentation is available at http://localhost:8000/docs when the application is running.

Total Development Time: 8-12 days

This implementation provides:

1. **Document processing** for PDFs, Word docs, PowerPoint, and text files
2. **AI-powered semantic search** using sentence embeddings
3. **Automatic summarization** using extractive techniques
4. **Simple web interface** for searching and uploading documents
5. **API for integration** with other applications

Advantages of This Approach:

- **100% Free**: Uses only open-source libraries
- **No external API dependencies**: Works offline
- **Lightweight**: Can run on modest hardware
- **Extensible**: Easy to add more features
- **Secure**: All data stays local

Limitations:

- Summarization quality is basic compared to commercial AI models
- Limited to text extraction (no image analysis)
- Requires manual document uploads
- No specialized domain knowledge without fine-tuning