ODF Interview Guide

Use this guide to structure your explanation of the **Offline Document Finder (ODF)** project during your interview.

1. The "Elevator Pitch" (30 Seconds)

"I built an **Offline Document Finder (ODF)**. It's a privacy-focused desktop tool that brings 'Google-like' semantic search to your local documents. Instead of just matching keywords like standard Windows search, it uses AI to understand the *meaning* of your query, so you can find files based on what they contain, even if you don't remember the exact filenames—all without your data ever leaving your computer."

2. The "Why" (The Problem)

Start by solving a relatable pain point. * **Context**: "We've all had that moment where we know we have a document about a specific topic (e.g., 'budget 2024'), but we can't find it because the file is named `final_v2.pdf` or `meeting_notes.docx`." * **Limitation**: "Traditional file search (like Windows Explorer) is brittle. It relies on **exact keyword matching**. If you search for 'invoice', it won't find a file named 'bill'." * **Privacy Gap**: "Cloud solutions like Google Drive solve this with AI, but they require uploading sensitive data. I wanted a solution that works **100% offline** for privacy and security."

3. The "How" (The Solution & Architecture)

Explain the tech stack to show your technical depth.

"I built a Python-based desktop application that consists of three main parts:"

A. The Document Processor (ETL Pipeline)

- "First, I built a pipeline to ingest documents."
- **Tech**: `pdfminer.six` (PDFs), `python-docx` (Word), standard I/O (Text).
- **Process**: "It scans a directory, extracts text from supported files, and cleans it for processing."

B. The AI Engine (The "Brain")

- "This is the core differentiator. I used **Sentence Transformers** (`sentence-transformers` library) to convert text into high-dimensional vector embeddings."
- "Instead of storing words, I store the *meaning* of the document as a list of numbers (a vector)."
- **Search**: "When a user types a query, I convert that query into a vector as well."

- **Tech**: I used **FAISS (Facebook AI Similarity Search)** by Meta. It's an optimized library for efficient similarity search. It calculates the distance between the query vector and document vectors to find the best semantic matches instantly."

C. The Interface

- "I wrapped this in a user-friendly GUI using **Python's Tkinter** so it's accessible to non-technical users."
- "The app runs locally, managing its own Python dependencies to ensure it works on any machine."

4. Key Talking Points (Impress Your Interviewer)

On Privacy & Security

"A key design decision was **Data Sovereignty**. By using local models (`faiss-cpu`, local embeddings), no user data ever touches the cloud. This makes it suitable for searching sensitive financial or legal documents."

On Performance

"I faced a challenge with speed. Running a deep learning model on a standard CPU can be slow. To solve this, I used `faiss-cpu`, which is highly optimized for vector calculations, ensuring search results appear in milliseconds even on a laptop."

On "Semantic" vs. "Keyword"

"The biggest value add is **synonym understanding**. If I search for 'car', ODF will find documents about 'automobiles' or 'vehicles' because they share semantic meaning in the vector space. Traditional regex search cannot do this."

5. Potential Questions You Might Get

"How does it handle large files?"

- *Answer*: "Currently, it processes the full text. For future improvements, I plan to implement 'chunking'—splitting large documents into smaller paragraphs to pinpoint exactly *where* inside the document the answer lies."

6. Code Execution Flow (Technical Deep Dive)

Use this if they ask "Walk me through the code flow".

A. The Setup (Initialization)

1. **`main.py`** is the entry point. It creates the `SearchWindow` (UI).

   `SearchWindow` initializes:
   - **`FileIndexer`**: The tool that reads files.
   - **`VectorSearch`**: The engine that manages the AI.
   - **`Embedder`**: Loads the AI model (e.g., `BAAI/bge-large-en-v1.5`) into memory.

B. The Indexing Process (When you click "Browse")

1. **User selects folder** -> `index_folder_async` starts a background thread (so UI doesn't freeze).

   `FileIndexer` crawls the folder:
   - Found `.pdf`? -> Uses `pdfminer.six` to extract text.
   - Found `.docx`? -> Uses `python-docx`.

   `VectorSearch` takes this raw text and **"Chunks"** it:
   - Splits long documents into smaller, meaningful segments (e.g., paragraphs).
4. **`Embedder`** converts these chunks into vectors.
5. **`FAISS`** builds a searchable index from these vectors and saves it to disk (`embeddings.index`).

C. The Search Process (When you type a query)

1. **User types query** -> `VectorSearch.search("query")` is called.
2. **Pre-processing**: The query is cleaned and normalized.
3. **Embedding**: The query is converted into a vector (numbers).
4. **Similarity Search**: FAISS compares the *query vector* against all *document-chunk vectors* to find the nearest neighbors (mathematical matches).
5. **Re-Ranking**: Results are refined (deduplicated and sorted by relevance score).
6. **Display**: The UI shows the top matches with a preview of the content.

Visual Diagram (Mental Model)

```
sequenceDiagram
   participant User
   participant UI as SearchWindow (UI)
   participant Indexer as FileIndexer
   participant Engine as VectorSearch
   participant AI as Embedder (Model)
   participant DB as FAISS Index

   Note over User, DB: 1. Indexing Phase
   User->>UI: Select Folder
   UI->>Indexer: Scan Folder
```

```
    Indexer->>UI: Return Text Content
    UI->>Engine: Build Index(Text)
    Engine->>AI: Generate Embeddings
    AI-->>Engine: Vectors
    Engine->>DB: Save Index to Disk

    Note over User, DB: 2. Search Phase
    User->>UI: Type "Budget 2024"
    UI->>Engine: Search("Budget 2024")
    Engine->>AI: Embed Query
    AI-->>Engine: Query Vector
    Engine->>DB: Find Nearest Neighbors
    DB-->>Engine: Top Results
    Engine-->>UI: Return Ranked Files
    UI-->>User: Show Results List
```

7. Specific Technical Questions (Deep Dives)

"How do you handle the folder input?"

Answer: "I used a two-part approach separating the UI from the logic:"

For the UI (User Interaction):

- "I used Python's built-in `tkinter.filedialog` module, specifically the `askdirectory()` function."
- *Why?*: "It triggers the **native OS folder picker**. This means on Windows, it looks like a Windows dialog, and on Mac, it looks like a Mac dialog. It handles permissions automatically and is familiar to the user."

For the Backend (File Traversal):

- "Once I get the path string, I use `os.walk()` in my `FileIndexer` class."

8. Future Roadmap (The "What's Next?" Question)

Show that you have product vision.

A. "Chat with your Documents" (RAG)

"The immediate next step is to add an LLM (like a small Llama model) on top of the search. Instead of just showing the *document*, the app would answer questions *from* the document. This turns it into a **RAG (Retrieval Augmented Generation)** system."

B. Hybrid Search

"Currently, I use pure semantic search. Sometimes, exact keyword matches (BM25) are better (e.g., searching for a specific invoice number 'INV-2024-001'). I plan to implement **Hybrid Search** to get the best of both: exact precision + semantic understanding."

C. OCR for Scanned PDFs

"Right now, ODF handles text-based PDFs. I want to integrate **Tesseract OCR** to handle scanned images and paperwork, which is a huge real-world use case."

D. Watching Folders

"Instead of manually re-indexing, I want to use the `watchdog` library to detect file changes in real-time and update the index automatically."