A Strategic and Technical Blueprint for the "Connecting the Dots" Hackathon

Introduction: From Static Pages to Intelligent Companions

The "Connecting the Dots" Hackathon presents a challenge that cuts to the core of modern document intelligence: transforming the Portable Document Format (PDF) from a static, passive container of information into a dynamic, interactive knowledge asset. The vision articulated in the challenge materials is not merely to parse documents but to build an experience where a PDF "spoke to you, connected ideas, and narrated meaning across your entire library".[1] This endeavor aims to reimagine reading itself, shifting the paradigm from simple content consumption to contextual understanding.

The central thesis of this report is that a winning solution will be defined not by the invention of novel artificial intelligence, but by superior engineering. The path to victory lies in the intelligent selection, meticulous integration, and high-performance optimization of existing, powerful tools within the tight constraints of the hackathon environment. The challenges are designed to test a team's ability to architect a robust, efficient, and compliant system that is greater than the sum of its parts.

A critical aspect of this architecture is the foundational relationship between the two rounds of the competition. Round 1A, the extraction of a structured document outline, is explicitly described as "the foundation for the rest of your hackathon journey".[1] This is not a trivial preliminary step; it is the bedrock upon which all subsequent intelligence is built. The accuracy and performance of the structural parser developed in Round 1A will directly determine the ceiling of success for the semantic analysis in Round 1B. A flawed or inefficient outline extractor will cripple the entire system, making it impossible to "connect the dots" in a meaningful way. Therefore, this report will first establish a definitive strategy for mastering Round 1A before building upon that success to architect the persona-driven intelligence engine for Round 1B.

Part I: Architecting the Document Structure Extractor (Round 1A)

The mission of Round 1A is to build a high-speed, high-accuracy system that ingests a raw PDF and outputs a structured JSON file containing the document's title and a hierarchical outline of its headings (H1, H2, H3).[1] Success in this round is contingent on meeting stringent performance and resource constraints, demanding a carefully considered technical approach.

1.1 Foundational Tooling: Selecting the Optimal PDF Parsing Engine

The choice of a core PDF parsing library is the single most consequential technical decision in Round 1A. This choice directly dictates the potential for accuracy, the feasibility of meeting the performance budget, and the ability to implement the sophisticated logic required to outperform simplistic approaches.

A comparative analysis of the most common Python libraries for PDF text extraction reveals a clear and decisive winner for this specific task.

- **PyPDF2/pypdf:** While popular for basic PDF manipulation tasks like merging or splitting pages, this family of libraries is fundamentally unsuited for the demands of this challenge. Its text extraction capabilities are known to struggle with complex layouts and, most critically, it does not provide reliable, granular access to the essential metadata—such as font names, sizes, weights, and precise positional coordinates—that is required for robust heading detection.[2] Attempting to build a high-accuracy solution on this library would be an exercise in futility.
- **PDFMiner:** This library represents a significant step up from PyPDF2, offering more robust parsing and the ability to access detailed information about text layout and font properties.[3] However, its primary drawback is performance. Multiple independent benchmarks have consistently shown that PDFMiner is substantially slower than its main competitor, PyMuPDF.[9] Given the strict execution time limit of 10 seconds for a 50-page document, this performance deficit introduces a significant and unnecessary risk.
- **PyMuPDF (Fitz):** This library is the unequivocally superior choice for this hackathon. It is a Python binding for MuPDF, a high-performance C library, which

makes it exceptionally fast. Benchmarks demonstrate that it can be orders of magnitude faster than other pure-Python libraries.[10] Its speed alone makes it the most compelling option under the tight time constraint. More importantly, PyMuPDF provides an extremely rich, structured data output via its page.get_text("dict") method. This method returns a nested dictionary detailing the page's content in a hierarchy of blocks, lines, and spans, where each element is annotated with its bounding box, and each span includes its font name, size, color, and flags (e.g., bold, italic).[7] This rich, structured metadata is the precise fuel required for an advanced, rule-based heading detection engine.

The constraints of Round 1A—specifically the 10-second execution limit and the sub-200MB model size—are not mere guidelines but hard architectural determinants. They strongly favor a lightweight, high-speed, rule-based approach over any machine learning model. A 50-page PDF must be processed in under 10 seconds, allocating a mere 200 milliseconds per page. The computational overhead of loading even a small ML model and performing inference on every potential text block within this time budget on a CPU is prohibitive. The problem must be reframed from an ML inference task to a high-performance algorithmic parsing challenge. PyMuPDF's combination of raw speed and detailed metadata output makes it the only logical and strategic choice to build a winning solution.[10]

Library	Execution Speed (Relative)	Output Granularity (Font/Bbox Data)	Dependency Footprint	Suitability for Challenge
PyMuPDF (Fitz)	Very High (Fastest)	Excellent (Blocks, lines, spans with size, font, flags, bbox)	Low (C-library binding)	**Optimal**
PDFMiner	Low to Medium	Good (Provides font/layout info)	Medium (Pure Python)	High Risk (Performance)
PyPDF2 / pypdf	Low (Slowest)	Poor (Lacks consistent, detailed metadata)	Low (Pure Python)	Unsuitable

1.2 The Logic of Hierarchical Structure Detection

A robust solution must heed the challenge's "Pro Tip": "Don't rely solely on font sizes for heading level determination".[1] PDFs in the wild are inconsistent, and a heuristic based on a single feature will fail. A durable solution requires a multi-factor scoring system built upon the rich data provided by PyMuPDF.

The core data structure for this logic is the dictionary returned by page.get_text("dict"). This dictionary organizes text into a hierarchy: the page contains blocks (paragraphs or images), blocks contain lines, and lines contain spans.[14] A span is a contiguous string of characters with identical font properties, making it the ideal atomic unit for analysis.

A multi-factor heuristic model can be implemented to classify each line of text. This involves two main phases: profiling the document's typography and then classifying lines based on their deviation from that profile.

1. Feature Extraction and Body Text Profiling:
   The first step is to iterate through the document to establish a statistical baseline for the "normal" body text. This creates a dynamic reference point that adapts to each document's unique styling, rather than relying on hardcoded assumptions. For each text span, the following features are extracted from the PyMuPDF dictionary output 7:
   - size: The font's point size.
   - font: The font's name (e.g., 'TimesNewRomanPS-BoldMT').
   - flags: An integer bitmask that indicates properties such as bold, italic, and monospace.[14]
   - bbox: The tuple (x_0, y_0, x_1, y_1) representing the bounding box coordinates.
   - text: The string content of the span.

   By analyzing the frequency of these properties across a sample of pages, the system can determine the most common font and size used for paragraph text. This becomes the document's "body text profile."
2. Heading Classification Rules:
   With the body text profile established, a second pass can classify each line. A line is a strong candidate for a heading if it exhibits a combination of the following characteristics:

   - **Relative Size:** Its font size is significantly larger than the baseline body text font size.
   - **Font Weight:** Its font name contains substrings like "Bold", "Black", "Heavy", or its flags property has the bold bit set.
   - **Typographic Style:** The text is rendered in ALL CAPS, a common convention for major headings.
   - **Spatial Position:** The line is typically left-aligned (indicated by a low x_0 value) and is often preceded by significant vertical whitespace. This can be inferred by measuring the vertical gap between the y_0 of the current block's bounding box and the y_1 of the previous block's bounding box.[18]
   - **Structural Patterns:** The text begins with a numerical or alphabetical pattern indicative of a heading, such as 1., 1.1., A., or Chapter 1. These can be efficiently detected using regular expressions.[19]
   - **Brevity:** The line is syntactically concise and does not form a complete, grammatical sentence.
3. Hierarchical Level Assignment (H1, H2, H3):
   Once a set of lines has been identified as headings, they must be assigned their hierarchical level. This can be achieved by clustering the headings based on their typographic properties.
   - **Primary Method (Typographic Clustering):** Group all identified headings by their unique combination of font size, font name, and weight. These clusters can then be sorted. The cluster with the largest font size typically corresponds to H1, the next distinct style to H2, and so on. This method is robust for documents with consistent styling.
   - **Secondary Method (Structural Analysis):** For documents that rely on numbering, the structure of the number itself can be used to determine hierarchy. A heading beginning with 2.1.3 is unambiguously an H3, subordinate to a preceding H2 (2.1) and H1 (2.). This can be used to refine or validate the results from typographic clustering.
4. Title Extraction:
   The document title is a special case of a heading. It is almost always located on the first page and is distinguished by having the largest font size in the entire document. It is also frequently centered on the page.

The following Python sketch illustrates the conceptual logic using PyMuPDF:

Python

```python
import fitz  # PyMuPDF
import json
import re

def extract_outline(pdf_path):
    doc = fitz.open(pdf_path)
    outline_data = {"title": "", "outline":}

    # Simplified logic for demonstration
    # A real implementation would profile body text first.

    # Find title (often largest text on first page)
    #... title extraction logic...

    # Process all pages for headings
    for page_num, page in enumerate(doc):
        text_dict = page.get_text("dict")
        blocks = text_dict.get("blocks",)
        for block in blocks:
            if block['type'] == 0: # Block contains text
                for line in block.get("lines",):
                    # Combine spans to get line text and dominant style
                    line_text = "".join([span['text'] for span in line['spans']])
                    if not line['spans']: continue

                    span = line['spans'] # Assume first span style is representative
                    font_size = span['size']
                    font_name = span['font']

                    # --- HEURISTIC RULES ---
                    is_bold = "bold" in font_name.lower()
                    is_short = len(line_text.split()) < 10
                    starts_with_number = re.match(r"^\d+(\.\d+)*", line_text.strip())

                    # Example rule: Large, bold, and short text is a heading
                    if font_size > 14 and is_bold and is_short:
                        level = "H1" # Default, needs refinement
                        if font_size < 20: level = "H2"
                if font_size < 16: level = "H3"

                outline_data["outline"].append({
                    "level": level,
                    "text": line_text.strip(),
                    "page": page_num + 1
                })

    return json.dumps(outline_data, indent=2)

# Usage:
# result_json = extract_outline("sample.pdf")
# print(result_json)
```

1.3 Achieving Peak Performance and Compliance

The proposed rule-based architecture is intrinsically designed to meet and exceed the performance and compliance requirements of Round 1A.

- **Performance Budget Compliance:** The 10-second limit for a 50-page PDF (averaging 200ms per page) is a strict constraint that the heuristic-based approach easily satisfies. The computational cost of the proposed logic is minimal. It primarily involves dictionary lookups, numerical comparisons, and string operations—all of which are computationally inexpensive and execute in microseconds. This stands in stark contrast to an ML-based approach, where model loading and inference would consume the majority of the time budget. The choice of PyMuPDF, the fastest available library, further ensures that the file I/O and initial parsing phase is as swift as possible.[1]
- **Model Size Compliance:** The solution has a negligible model footprint. The core logic is algorithmic, not model-based. The only significant dependency is the PyMuPDF library itself. As there is no ML model to load, the ≤200MB constraint is met by a vast margin, effectively eliminating it as a concern.[1]
- **Offline Operation:** The entire pipeline is self-contained. It relies on the PyMuPDF library and standard Python libraries, none of which require network access during execution. This directly fulfills the "no network/internet calls" requirement, ensuring the solution will run correctly in the isolated judging environment.[1]

1.4 Securing Bonus Points: A Pragmatic Approach to Multilingual Handling

The challenge offers bonus points for handling non-Latin languages, specifically citing Japanese as an example.[1] A common but incorrect impulse would be to integrate an Optical Character Recognition (OCR) engine like Tesseract to handle this requirement. This path is a trap that would lead to a slow, bloated, and error-prone solution.

OCR is a technology for *recognizing* text from images. It is only necessary for scanned PDFs where the pages are images rather than encoded text. For digitally-born PDFs, which are the focus of this hackathon, the text data already exists within the file structure; the challenge is one of correct *extraction* and *decoding*, not recognition.[2] Furthermore, OCR engines are slow, require large language data packs that would violate the size constraint, and introduce a layer of recognition errors that would compromise accuracy.[21]

The correct and far more elegant strategy is to leverage the native capabilities of the chosen toolchain. PyMuPDF has excellent built-in support for CJK (Chinese, Japanese, Korean) fonts and character encodings.[17] When processing a digitally-born PDF containing Japanese text,

page.get_text("dict") will correctly extract the Japanese characters along with their associated font and positional metadata.[24]

The multi-factor heading detection logic proposed in section 1.2 is fundamentally language-agnostic. It relies on relative typographic and spatial cues, not the semantic content of the text itself:

- A heading in a Japanese document is still likely to have a larger font size than the body text.
- It is still likely to use a bold-weighted font.
- It is still likely to be positioned with extra whitespace.

Therefore, the same heuristic model will work on extracted Japanese text without any modification. This approach is highly efficient, requires no additional libraries or data files, maintains peak performance, and stays well within all resource constraints, making it the optimal strategy to secure the bonus points.

Part II: Building the Persona-Driven Intelligence Engine (Round 1B)

Round 1B elevates the challenge from structural parsing to semantic understanding. The mission is to build an "intelligent document analyst" that can extract and prioritize the most relevant sections from a collection of documents based on a user persona and their job-to-be-done.[1] The constraints are relaxed compared to 1A (≤1GB model size, ≤60s processing time), explicitly creating space for a lightweight machine learning solution.[1]

2.1 From Structure to Semantics: The Core Intelligence Architecture

The most robust, scalable, and generalizable architecture for this task is one based on **semantic vector search**. This approach moves beyond fragile keyword matching and captures the underlying meaning—the semantic intent—of both the user's query and the document content. By representing text as numerical vectors (embeddings) in a high-dimensional space, relevance can be calculated as proximity in that space.[25]

The proposed data flow for the intelligence engine is as follows:

1. **Input Processing:** The system receives a collection of 3-10 PDF files, a persona description (e.g., "PhD Researcher in Computational Biology"), and a job-to-be-done (e.g., "Prepare a comprehensive literature review focusing on methodologies, datasets, and performance benchmarks").[1]
2. **Semantic Chunking:** For each PDF, the system uses the structured JSON outline generated by the Round 1A solution to segment the document into meaningful semantic units or "chunks." A logical definition for a chunk is a heading (H1, H2, or H3) combined with all the text that follows it, up to the next heading of an equal or higher level. This method ensures that chunks represent coherent sections of the document, far superior to arbitrary fixed-size splitting.
3. **Query Formulation:** The persona and job-to-be-done strings are concatenated into a single, rich query string. For the example above, the query would be: "As a PhD Researcher in Computational Biology, I need to prepare a comprehensive literature review focusing on methodologies, datasets, and performance

benchmarks." This combined query provides a wealth of semantic context for the model.

4. **Vector Embedding:** A pre-trained sentence-transformer model is used to perform the core task of semantic encoding. It converts the formulated query string into a query vector. It also iterates through all the semantic chunks from all documents, converting each one into a corresponding chunk vector.
5. **Relevance Ranking:** The relevance of each chunk to the user's need is determined by calculating the cosine similarity between the query vector and every chunk vector. Cosine similarity measures the orientation (or angle) between two vectors, providing a normalized score of their semantic relatedness, independent of their magnitude.
6. **Output Generation:** The chunks are ranked in descending order based on their calculated cosine similarity scores. The top-ranked chunks are the most relevant sections. This ranked list is then used to construct the final JSON output in the precise format specified by the challenge, including metadata, Extracted Section with importance_rank, and Sub-section Analysis.[1]

2.2 Selecting the Ideal On-Device Embedding Model

The success of the semantic search architecture hinges on the selection of an appropriate embedding model. The ideal model must satisfy several competing requirements: it must be highly effective at capturing semantic meaning, small enough to comply with the ≤1GB size limit, fast enough for efficient CPU-only inference, and capable of operating entirely offline.

The sentence-transformers library is the de facto standard for this type of task. It provides a simple, unified interface for using thousands of pre-trained models that are optimized for generating high-quality embeddings for sentences, paragraphs, and documents.[26]

Within the vast landscape of available models, **sentence-transformers/all-MiniLM-L6-v2** emerges as the optimal choice for this hackathon's specific constraints.

- **Compact Size:** The model is approximately 80-90MB on disk, consuming less than 10% of the 1GB budget. This leaves ample room for the operating system, Python runtime, and other libraries.[30]

- **High Performance:** As a "MiniLM" (Mini Language Model), it is designed for efficiency. It is exceptionally fast on CPU, which is critical for meeting the 60-second processing time limit for a collection of up to 10 documents.[30]
- **Proven Effectiveness:** Despite its small size, all-MiniLM-L6-v2 is a powerful, general-purpose model trained on a massive dataset of over 1 billion text pairs. It produces 384-dimensional embeddings that are highly effective for semantic search, clustering, and other similarity tasks, making it a reliable and well-regarded choice in the NLP community.[30]

Model Name	Size (MB)	Embedding Dimensions	Relative CPU Speed	General Performance
all-MiniLM-L6-v2	~86	384	Very High	Excellent
all-MiniLM-L12-v1	~120	384	High	Excellent
multi-qa-MiniLM-L6-cos-v1	~86	384	Very High	Specialized for Q&A

As the table illustrates, all-MiniLM-L6-v2 provides the best overall balance of size, speed, and general-purpose performance, making it the lowest-risk, highest-reward choice for this challenge.[26] While a model like

multi-qa-MiniLM-L6-cos-v1 is optimized for question-answering, the persona + job-to-be-done format is more of a descriptive statement than a direct question, making the general-purpose all-MiniLM a more suitable fit.

2.3 Implementing High-Relevance Semantic Ranking

The implementation of the semantic ranking pipeline can be broken down into a series of distinct, manageable steps. For the final ranking, a Cross-Encoder model is recommended for its superior accuracy in re-ranking a small set of candidate documents against a query.[29]

1. Offline Model Loading:
   To ensure offline compliance, the sentence-transformer and cross-encoder

models must be downloaded during the Docker image build process and loaded from a local directory during execution.

```python
from sentence_transformers import SentenceTransformer, CrossEncoder, util

# During build: download and save models.
# During runtime: load from a local path.
embedding_model_path = "./models/all-MiniLM-L6-v2"
reranker_model_path = "./models/cross-encoder-ms-marco-MiniLM-L6-v2"

embedder = SentenceTransformer(embedding_model_path)
reranker = CrossEncoder(reranker_model_path)
```

2. Document Chunking and Query Formulation:
   A function is needed to parse the 1A-generated JSON and the raw PDF text into a list of structured chunk objects.

```python
def create_chunks(pdf_path, outline_json):
    # Logic to read PDF and use outline_json to segment text
    # into chunks, where each chunk is a dict:
    # {'doc': pdf_path, 'page': 5, 'title': 'Section Title', 'content': '...'}
    chunks =
    #... implementation...
    return chunks

persona = "Investment Analyst"
job = "Analyze revenue trends, R&D investments, and market positioning strategies"
query = f"Persona: {persona}. Job: {job}"
```

3. Initial Retrieval with Embedding Model:
   First, use the fast embedding model to retrieve a larger set of potentially relevant documents from the entire collection. This is a "retrieval" step.

```python
# Assume 'all_chunks' is a list of all chunks from all documents
chunk_contents = [chunk['content'] for chunk in all_chunks]

query_embedding = embedder.encode(query, convert_to_tensor=True)
chunk_embeddings = embedder.encode(chunk_contents, convert_to_tensor=True)

# Compute cosine similarities
```

```python
cosine_scores = util.cos_sim(query_embedding, chunk_embeddings)

# Get top-k candidates for re-ranking (e.g., top 50)
top_k = 50
top_results = torch.topk(cosine_scores, k=min(top_k, len(all_chunks)))

candidate_chunks = [all_chunks[i] for i in top_results.indices]
```

4. Final Ranking with Cross-Encoder:
   Now, use the more accurate (but slower) Cross-Encoder to re-rank the smaller set of candidate chunks. This is the "re-rank" step.
   Python
   ```python
   # Prepare pairs for the reranker
   reranker_pairs = [[query, chunk['content']] for chunk in candidate_chunks]

   # Predict scores
   reranker_scores = reranker.predict(reranker_pairs)

   # Combine scores with chunk info and sort
   for i in range(len(candidate_chunks)):
       candidate_chunks[i]['score'] = reranker_scores[i]

   ranked_chunks = sorted(candidate_chunks, key=lambda x: x['score'], reverse=True)
   ```

5. Formatting the Final Output:
   The final step is to iterate through the ranked_chunks and construct the JSON output according to the specified schema, assigning an importance_rank based on the sorted order.
   Python
   ```python
   output_json = {
       "metadata": {... },
       "extracted_section":,
       "sub-section_analysis": # Populate as required
   }

   for i, chunk in enumerate(ranked_chunks):
       output_json["extracted_section"].append({
           "document": chunk['doc'],
           "page_number": chunk['page'],
           "section_title": chunk['title'],
   ```

```
        "importance_rank": i + 1
    })
    # Populate sub-section analysis if needed
```

This two-stage retrieve-and-rerank process combines the speed of embedding models for broad searching with the accuracy of cross-encoders for fine-grained final ranking, providing a state-of-the-art solution.

2.4 Designing for Generalization

A key requirement of the Round 1B challenge is that the solution must be "generic to generalize to this variety" of document domains, personas, and jobs.[1] The proposed semantic vector search architecture is inherently designed for this generalization.

The power of this approach lies in the nature of the sentence-transformer model itself. all-MiniLM-L6-v2 was pre-trained on an enormous and diverse corpus of over one billion text pairs drawn from various corners of the internet.[30] This training process forces the model to learn a rich, latent representation of language that captures semantic meaning far beyond simple keywords.

When the model embeds the query "Analyze revenue trends" and a document section discussing "year-over-year growth in net income," it maps them to nearby points in the vector space because it understands their semantic relationship, even though they share no keywords. This capability is domain-agnostic. The model does not need to be explicitly trained on financial reports to understand financial concepts, or on scientific papers to understand scientific methodologies.

Because the system operates on these learned semantic representations, it contains no hardcoded logic, no domain-specific keyword lists, and no rules tailored to a particular type of document. It will perform its function equally well on the provided test cases—academic research, business analysis, and educational content—and on any other domain it might encounter. This inherent flexibility and robustness are the hallmarks of a well-architected semantic system and directly address the core demand for a generalizable solution.

Part III: Engineering a Compliant, Production-Ready Solution

The final and most critical phase of the hackathon is packaging the developed logic into a compliant, executable, and well-documented submission. Excellence in software engineering practices at this stage is as important as the algorithmic cleverness of the preceding parts.

3.1 Crafting the Optimal Offline Docker Container

The submission must be a self-contained Docker image that runs on a linux/amd64 architecture, executes entirely offline, and processes all PDFs from a mounted input directory to a mounted output directory.[1] The professional standard for creating an optimized and secure image for this purpose is a

multi-stage Dockerfile. This approach separates the build-time environment from the final runtime environment, resulting in a minimal, clean, and more secure final image.[32]

The proposed Dockerfile will consist of two primary stages:

1. The builder Stage:
   This stage is used to prepare all necessary artifacts.
   - It starts from a full-featured base image (e.g., python:3.10-bookworm) that includes common build tools like git and curl.
   - It creates a dedicated directory for the ML models. A small Python script or curl command is used to download the all-MiniLM-L6-v2 and cross-encoder/ms-marco-MiniLM-L-6-v2 model files from the Hugging Face Hub and save them into this directory.
   - It copies the requirements.txt file and installs all Python package dependencies (e.g., pymupdf, sentence-transformers, torch) into a self-contained virtual environment, for example, at /opt/venv. This isolates dependencies and makes them easy to transfer to the final stage.[33]
2. The final Stage:
   This stage constructs the lean, production-ready image.
   - It begins with a minimal base image, explicitly specifying the target platform: FROM --platform=linux/amd64 python:3.10-slim-bookworm.[1] The

  -slim variant lacks many non-essential packages, significantly reducing the final image size and security attack surface.[36]
  - It creates a non-root user and group for executing the application. Running as a non-root user is a critical security best practice that limits the potential impact of a container breakout.[33]
  - It copies the pre-built artifacts from the builder stage: the Python virtual environment (/opt/venv) and the downloaded models directory are copied over.
  - It copies the application source code into the working directory.
  - It sets the PATH environment variable to include the virtual environment's bin directory, so that the Python interpreter and installed packages are used.
  - It sets the USER to the newly created non-root user.
  - Finally, it defines the CMD or ENTRYPOINT to launch the main Python script that orchestrates the entire process.

The entrypoint script (main.py) is the conductor of the container. It will be designed to:

- Use Python's os or pathlib module to scan the /app/input directory for all files ending with .pdf.[37]
- Loop through each discovered PDF file.
- For each PDF, execute the full pipeline: run the Round 1A logic to generate the structured outline, then feed this and the other documents into the Round 1B engine to produce the final persona-driven analysis.
- Write the final JSON output to a file with the corresponding name (e.g., sample.pdf -> sample.json) in the /app/output directory, as specified by the execution requirements.[1]

A complete, annotated Dockerfile implementing this strategy would look as follows:

Dockerfile

```
# Stage 1: The Builder
# Use a full-featured image to download models and install packages
FROM python:3.10-bookworm as builder

# Install tools needed for downloading
RUN apt-get update && apt-get install -y curl git
```

```dockerfile
# Set up a virtual environment
ENV VENV_PATH=/opt/venv
RUN python3 -m venv $VENV_PATH
ENV PATH="$VENV_PATH/bin:$PATH"

# Install sentence-transformers which will cache models
# We do this in a separate layer to leverage Docker caching
RUN pip install sentence-transformers==2.2.2
# Pre-download and cache the required models
RUN python -c "from sentence_transformers import SentenceTransformer, CrossEncoder; \
   SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2'); \
   CrossEncoder('cross-encoder/ms-marco-MiniLM-L-6-v2')"

# Copy requirements and install all dependencies into the venv
COPY requirements.txt.
RUN pip install --no-cache-dir -r requirements.txt

# ---

# Stage 2: The Final Image
# Use a slim, platform-specific base image for a small footprint
FROM --platform=linux/amd64 python:3.10-slim-bookworm

# Set working directory
WORKDIR /app

# Create a non-root user for security
RUN useradd --create-home --shell /bin/bash appuser
USER appuser

# Copy the virtual environment from the builder stage
ENV VENV_PATH=/opt/venv
COPY --from=builder $VENV_PATH $VENV_PATH
ENV PATH="$VENV_PATH/bin:$PATH"

# Copy the cached models from the builder stage
COPY --from=builder /root/.cache/torch/sentence_transformers
/home/appuser/.cache/torch/sentence_transformers

# Copy application source code
COPY --chown=appuser:appuser./src./src

# Set the entrypoint for the container
```

```
CMD ["python", "src/main.py"]
```

3.2 Final Submission Strategy and Documentation

The final submission is not just code; it is a complete project that includes documentation and is organized in a professional manner.

- **Git Repository Structure:** A clean and logical project structure is recommended for clarity and maintainability.
  - Dockerfile: Located in the project root.
  - requirements.txt: Lists all Python dependencies with pinned versions.
  - .dockerignore: Excludes unnecessary files (.git, __pycache__, local test data) from the Docker build context.
  - src/: A directory containing all Python source code (main.py, round1a.py, round1b.py, etc.).
  - README.md: The primary documentation file.
- **Authoring a High-Quality README.md:** The README.md is a scored deliverable and must be treated as such. It should be clear, concise, and comprehensive, explaining the following points as required by the submission checklist [1]:
  - **Your Approach:** A high-level summary of the technical strategy. This should explain the choice of a rule-based heuristic engine for Round 1A (leveraging PyMuPDF for speed and metadata richness) and a two-stage semantic vector search (retrieve and re-rank) for Round 1B.
  - **Models or Libraries Used:** An explicit list of the key dependencies and their roles:
    - PyMuPDF (Fitz): For high-performance PDF parsing and metadata extraction.
    - sentence-transformers: For semantic embedding and ranking.
    - sentence-transformers/all-MiniLM-L6-v2: The embedding model for initial retrieval.
    - cross-encoder/ms-marco-MiniLM-L-6-v2: The Cross-Encoder model for final re-ranking.
    - PyTorch: As a backend for the transformer models.
  - **How to Build and Run Your Solution:** While the judges will use the prescribed commands, providing clear, copy-pasteable build and run instructions demonstrates a thorough understanding of the containerization process and professionalism.

- **Final Submission Checklist:** Before submission, a final review against the checklist is mandatory.[1] This includes verifying that the Git repository is private until the specified deadline, that all dependencies are correctly installed within the container, and that there are absolutely no hardcoded file-specific logic or external API/web calls in the final code. This meticulous final check ensures that the submission is fully compliant and ready for judging.

Works cited

1. adobe 2nd round.pdf
2. Extract Text from a PDF — PyPDF2 documentation - Read the Docs, accessed July 16, 2025, https://pypdf2.readthedocs.io/en/3.0.0/user/extract-text.html
3. A Guide to PDF Extraction Libraries in Python - Metric Coders, accessed July 16, 2025, https://www.metriccoders.com/post/a-guide-to-pdf-extraction-libraries-in-python
4. Data Extraction from Unstructured PDFs - Analytics Vidhya, accessed July 16, 2025, https://www.analyticsvidhya.com/blog/2021/06/data-extraction-from-unstructured-pdfs/
5. Extract text from PDF File using Python - GeeksforGeeks, accessed July 16, 2025, https://www.geeksforgeeks.org/python/extract-text-from-pdf-file-using-python/
6. Extracting Information from PDF Documents with Python | by Saurabh Chodvadiya - Medium, accessed July 16, 2025, https://medium.com/@chodvadiyasaurabh/extracting-information-from-pdf-documents-with-python-25b3fa24c2e5
7. How to extract text from PDF on the basis of font size using python ..., accessed July 16, 2025, https://medium.com/@i190712/how-to-extract-text-from-pdf-on-the-basis-of-font-size-using-python-libraries-e931e583749b
8. PDFDataExtractor: A Tool for Reading Scientific Text and Interpreting Metadata from the Typeset Literature in the Portable Document Format, accessed July 16, 2025, https://pmc.ncbi.nlm.nih.gov/articles/PMC9049592/
9. Features Comparison - PyMuPDF 1.26.3 documentation, accessed July 16, 2025, https://pymupdf.readthedocs.io/en/latest/about.html
10. Comparing 4 methods for pdf text extraction in python | by Jeanna Schoonmaker | Social Impact Analytics | Medium, accessed July 16, 2025, https://medium.com/social-impact-analytics/comparing-4-methods-for-pdf-text-extraction-in-python-fd34531034f
11. Which is faster at extracting text from a PDF: PyMuPDF or PyPDF2? : r/learnpython - Reddit, accessed July 16, 2025, https://www.reddit.com/r/learnpython/comments/11ltkqz/which_is_faster_at_extracting_text_from_a_pdf/
12. Text - PyMuPDF 1.26.3 documentation, accessed July 16, 2025,

https://pymupdf.readthedocs.io/en/latest/recipes-text.html
13. Tutorial - PyMuPDF 1.26.3 documentation, accessed July 16, 2025, https://pymupdf.readthedocs.io/en/latest/tutorial.html
14. Appendix 1: Details on Text Extraction - PyMuPDF 1.26.3 documentation, accessed July 16, 2025, https://pymupdf.readthedocs.io/en/latest/app1.html
15. How to Extract Text from a PDF Using PyMuPDF and Python | by Neurond AI | Medium, accessed July 16, 2025, https://neurondai.medium.com/how-to-extract-text-from-a-pdf-using-pymupdf-and-python-caa8487cf9d
16. A Comparative Study of PDF Parsing Tools Across Diverse Document Categories - arXiv, accessed July 16, 2025, https://arxiv.org/pdf/2410.09871
17. Font - PyMuPDF 1.26.3 documentation, accessed July 16, 2025, https://pymupdf.readthedocs.io/en/latest/font.html
18. Is there a way to delete headers/footers in PDF documents? #2259 - GitHub, accessed July 16, 2025, https://github.com/pymupdf/PyMuPDF/discussions/2259
19. How to extract text under specific headings from a pdf? - Stack Overflow, accessed July 16, 2025, https://stackoverflow.com/questions/48107611/how-to-extract-text-under-specific-headings-from-a-pdf
20. Extract Text from Images and Scanned PDFs with Python (OCR) | by Alice Yang - Medium, accessed July 16, 2025, https://medium.com/@alice.yang_10652/extract-text-from-images-and-scanned-pdfs-with-python-2087cb1e0a7b
21. How to properly extract Japanese txt from PDF files - Stack Overflow, accessed July 16, 2025, https://stackoverflow.com/questions/71224718/how-to-properly-extract-japanese-txt-from-pdf-files
22. JP OCR Language not working! - API - OpenAI Developer Community, accessed July 16, 2025, https://community.openai.com/t/jp-ocr-language-not-working/948829
23. Mastering PDF Text with PyMuPDF's 'insert_htmlbox': What You Need to Know - Medium, accessed July 16, 2025, https://medium.com/@pymupdf/mastering-pdf-text-with-pymupdfs-insert-htmlbox-what-you-need-to-know-705a044f07f3
24. How to extract clean japanese text from the pdf folder in python - Stack Overflow, accessed July 16, 2025, https://stackoverflow.com/questions/79102087/how-to-extract-clean-japanese-text-from-the-pdf-folder-in-python
25. Extracting Structured Data from PDFs Using AI, Python, and Vector Databases., accessed July 16, 2025, https://padulaguruge.medium.com/extracting-structured-data-from-pdfs-using-ai-python-and-vector-databases-dbd2c310b0db
26. Using Sentence Transformers at Hugging Face, accessed July 16, 2025, https://huggingface.co/docs/hub/sentence-transformers
27. Semantic Search with Hugging Face Models and Datasets - SingleStore Spaces,

accessed July 16, 2025, https://www.singlestore.com/spaces/semantic-search-with-hugging-face-models-and-datasets/
28. Sentence Transformers - Hugging Face, accessed July 16, 2025, https://huggingface.co/sentence-transformers
29. sentence-transformers·PyPI, accessed July 16, 2025, https://pypi.org/project/sentence-transformers/
30. sentence-transformers/all-MiniLM-L6-v2 · Hugging Face, accessed July 16, 2025, https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2
31. sentence-transformers/all-MiniLM-L12-v1 - Hugging Face, accessed July 16, 2025, https://huggingface.co/sentence-transformers/all-MiniLM-L12-v1
32. Offline, Multistage Python Dockerfile - DEV Community, accessed July 16, 2025, https://dev.to/alimehr75/offline-multistage-python-dockerfile-n2j
33. How to Dockerize a Python Application | by Leonardo Rodrigues Martins, accessed July 16, 2025, https://python.plainenglish.io/how-to-create-a-docker-image-of-a-python-application-for-production-b5705a36dcbf
34. python - Official Image - Docker Hub, accessed July 16, 2025, https://hub.docker.com/_/python
35. amd64/python - Docker Image, accessed July 16, 2025, https://hub.docker.com/r/amd64/python/
36. The best Docker base image for your Python application (May 2024), accessed July 16, 2025, https://pythonspeed.com/articles/base-image-python-docker-images/
37. How to Move Files with Python - YouTube, accessed July 16, 2025, https://www.youtube.com/watch?v=Q3loeTkYA8Q
38. Python: How to List Files in Directory | Built In, accessed July 16, 2025, https://builtin.com/data-science/python-list-files-in-directory
39. How to iterate over files in directory using Python? - GeeksforGeeks, accessed July 16, 2025, https://www.geeksforgeeks.org/python/how-to-iterate-over-files-in-directory-using-python/
40. Tutorial: Iterate Over Files in a Directory Using Python - Pierian Training, accessed July 16, 2025, https://pieriantraining.com/iterate-over-files-in-directory-using-python/