AI Document Q&A System

Mando: Automating enterprise system integration and support

Executive Summary

Welcome to the **General-Purpose Document Question Answering System** hackathon challenge! In this project, you will develop an AI-powered question-answering tool that can ingest documents in a wide range of formats and accurately answer users' questions based on their content. The goal is to create a **universal Q&A system** capable of understanding PDFs, Word documents, presentations, spreadsheets, images, and more. This reflects a real-world need to pull knowledge from diverse sources and formats, making information retrieval seamless and efficient.

In this challenge, your system will accept uploads of files (PDF, DOCX, PPTX, XLSX, PNG, JPG, CSV, JSON, TXT) and build a knowledge base from their contents. Users can then ask natural language questions, and your system should return correct, contextually relevant answers derived from the documents. You are encouraged to leverage modern techniques – for example, combining Large Language Models (LLMs) with document parsing and search – to achieve high accuracy. Creativity in approach, **robust handling of edge cases**, and a focus on fidelity (i.e. answers that truly reflect the source documents) will be key evaluation criteria. This project is an opportunity to showcase your skills in natural language processing, data handling, and integration of multiple technologies into one cohesive application. We encourage you to think outside the box and have fun with the implementation!

Technical Specifications

Your question-answering system should meet the following technical specifications and features:

- **Multi-Format File Support:** The system must support uploading and processing of various document types, including **PDF**, **DOCX** (Word documents), **PPTX** (PowerPoint presentations), **XLSX** (Excel spreadsheets), **CSV** (comma-separated values files), **JSON** (structured data files), **PNG/JPG** images, and plain text files. For each file, the relevant content should be extracted and indexed for search. For

example, PDFs, Word docs, and PowerPoints contain text that can be parsed; spreadsheets contain tables of data that might need conversion to a queryable format; images (including scanned documents in PDF or image form) may contain text that needs OCR extraction. You should use reliable parsing libraries or tools for each format (e.g. PDF text extraction libraries, Office document parsers, etc.) to ingest the content. The end result is that a user can upload any combination of these files and the system will incorporate all their content into its knowledge store.

- **Accurate Question Answering:** Given an arbitrary natural-language question from the user, the system should provide a correct and well-formed answer based *solely on the content of the uploaded documents*. This means the system needs to effectively search the gathered content for relevant information and formulate an answer without hallucinating or introducing unsupported facts. It should be able to handle direct fact-based questions (e.g. "What is the definition of X as described in the PDF?") as well as slightly more complex queries (e.g. "Summarize the main findings from the report" or "Based on the data, what was the sales trend in 2020?"). The answering process may involve a combination of information retrieval and using an LLM or algorithm to compose the answer. Importantly, the system should be able to search **across all uploaded documents** – if the answer is spread over multiple files or if one document references information in another, the system should still find it. Answers should be presented in clear English sentences and should **preserve the original meaning and facts** from the sources (high fidelity). If the question can't be answered from the documents, the system should respond gracefully (e.g. "The information is not available in the provided documents."). High accuracy and fidelity in answers are crucial for a top-performing solution.
- **Reference Link Crawling:** Many documents include references or hyperlinks to external web pages. Your system should handle this by detecting references (for example, a PDF might cite a URL or a DOCX might contain a hyperlink) and attempting to retrieve the content from those links as an **additional knowledge source**. This effectively means your question-answering is not limited only to the uploaded files but can extend to one-hop web crawling. For instance, if a PowerPoint slide has a link to an online article for further reading, or a PDF references a web URL for full details, your system should fetch that webpage's text and include it in its corpus for searching. This extra content can then be used to answer questions that the original documents alone couldn't. Ensure that the crawling is done responsibly (e.g., fetch only the page that's linked, not an entire website crawl) and handle cases where the link might be broken or unreachable (perhaps skip or warn gracefully). By incorporating referenced material, your system demonstrates thoroughness in

finding answers. *(Hint: You can use standard libraries or HTTP clients to fetch HTML content, and then parse text from HTML. Be mindful of stripping out navigation or irrelevant parts).*

- **Image OCR Integration:** Besides text files, the system must handle images that contain text. This includes both standalone image files (PNG, JPG scans or photos of documents) and images embedded inside PDFs or slides. To achieve this, integrate an **Optical Character Recognition (OCR)** component into your pipeline. When an image file is uploaded (or an image is encountered inside a document), the system should run OCR to extract any text contained in that image. The extracted text should then be treated like any other text content – indexed and made searchable for answering questions. This is especially important for scanned documents or photographs of text, where the content isn't digitally accessible to begin with. You are free to use external OCR APIs or libraries (for example, **Tesseract OCR** or Google Vision API) rather than writing your own OCR from scratch (in fact, using a well-tested OCR engine is **strongly encouraged** to save time and improve accuracy). Keep in mind that OCR might occasionally mis-recognize characters, so consider strategies to clean or validate OCR output if possible. By doing OCR on images, your system ensures no information is missed just because it was in a picture. **Edge case:** Even PDFs or PPTXs can sometimes be just scanned images of text; your system should detect if a page has no extractable text and fallback to OCR for those pages.
- **Semantic Search and Retrieval:** Implement a robust search capability that goes beyond simple keyword matching. Users might phrase questions differently than the text in the document. For example, a document might say "Annual revenue grew 5% year-over-year," and the user asks "How much did revenue increase last year?" – a keyword search for "increase" might fail, but a semantic search would recognize "grew 5%" is related. To handle this, use **semantic similarity** techniques to retrieve relevant document snippets. This typically involves embedding the document text into vector representations and likewise embedding the query, then finding which documents or paragraphs have vectors most similar to the query vector. Semantic search will help the system find the right answer even if there isn't an exact keyword overlap. You can leverage existing embedding models or APIs to generate these embeddings. Many teams use pre-trained models (like Sentence Transformers or OpenAI embeddings) and a vector database or index (FAISS, Chroma, etc.) to enable fast similarity search. The bottom line: ensure that if the answer is present in the documents (even phrased differently), your retrieval mechanism can surface it. A combination of keyword indexing and semantic embeddings (a **hybrid search**) could

also be a good approach for robustness. This component greatly improves the intelligence of your Q&A system by adding an understanding of context and meaning.

- **Structured Data Handling:** Not all answers are in free-text paragraphs – some may reside in structured data like tables or JSON records. The system should be capable of dealing with **tabular or structured data files** (CSV, XLSX, JSON) in a way that allows business intelligence style questions. This means if a user asks a question like "What was the total sales for 2022?" and one of the uploaded files is a spreadsheet or CSV containing sales data by year, the system should be able to interpret the question, look into the data, and compute or retrieve the answer. One approach is to convert such files into a **programmatic format** (e.g., load CSV/XLSX into a DataFrame or database table) and then either query it directly or let an LLM reason over it. You might, for example, use Python's pandas to load a CSV, and if a question seems related to that data, perform a calculation and then respond with the result. Alternatively, you could translate the table into a textual summary or use tools that allow natural language queries over data. The key is that numbers and facts buried in rows/columns should not be ignored. Your system should handle basic operations like summation, averaging, filtering of data if needed to answer a question. This is an opportunity to integrate a mini "data analytics" component into your QA system. Keep the scope reasonable (we don't expect a full SQL engine for every possible complex query), but aim to cover typical use cases like totals, counts, or finding specific entries from the structured data. This will show that your solution is **general-purpose** not only for unstructured text but also for structured information.

In summary, the Technical Specifications boil down to building a pipeline that can **ingest heterogeneous document formats**, **extract all relevant content (text, data, images)**, **index that content for intelligent search**, and **accurately answer questions** by pulling evidence from this content. You are free to design the architecture as you see fit – for example, you might build a document processing module and a separate Q&A module that uses the processed data. Using Python for the core implementation is recommended, given the rich ecosystem of libraries for file parsing and NLP, but you may include other languages/components if needed (with justification). Ensure that the system is reasonably efficient in handling documents (it should handle a few lengthy documents without timing out) and is reliable when faced with the edge cases described. We want to see a **comprehensive, working solution** that proves your team can integrate multiple technologies to solve a complex problem.

Requirements

To successfully complete the project, please make sure your submission meets the following requirements:

- **Allowed Technologies:** You are encouraged to use **Python** as the main programming language for this project due to its extensive libraries for NLP and data processing. You are **allowed (and encouraged) to leverage pre-built tools, frameworks, and APIs** to accomplish the tasks. This includes using Large Language Model APIs or libraries (for example, OpenAI GPT-4, Hugging Face transformers) for understanding questions or generating answers, using OCR engines (Tesseract, Google Vision API, etc.) for text extraction from images, and using embedding or vector database services for semantic search. There is no requirement to train your own models from scratch – you can use off-the-shelf models and services to focus on integration and application. In fact, writing a custom OCR or training a brand-new NLP model is discouraged given the time constraints; instead, use existing state-of-the-art tools to your advantage. Make sure to comply with any free-tier or academic usage limits of external APIs if you choose to use them. If you use cloud services or APIs, document them and provide necessary configuration or keys (privately, in environment variables) so that judges can run your code. **Note:** While Python is recommended, if you have a strong reason to use another language for certain components (say, a JS front-end or a Java OCR tool), that's acceptable – just ensure everything ties together and is well-documented.
- **Code Submission (GitHub Repo):** Your team must submit a **self-contained GitHub repository** that contains all source code, documentation, and instructions to run your project. The repo should have a clear README.md that describes the project, how to install any dependencies, how to run the application or tests, and any other notes for the judges. Organize your code logically (using folders/modules for front-end, back-end, models, etc., as appropriate) and include comments to explain complex sections. We will be looking at code readability and organization as part of the evaluation, so write clean, well-structured code. All required libraries or models should be listed (perhaps in a requirements.txt or environment YAML for Python). If you utilize large language model APIs or other cloud services, explain in the README how to set those up (but avoid committing sensitive API keys in the repo!). Essentially, the GitHub repo should contain everything needed for someone else to understand and run your project.

- **Deployed Demo:** In addition to the code repository, you need to provide a **deployed, working demo** of your system accessible via a URL. This could be a web application or an API endpoint. The easiest formats would be either a simple web UI where a user can upload documents and ask questions, or a RESTful API where we can send documents and queries and get answers. You can deploy the demo on any platform of your choice (for example, Heroku, GitHub Pages (if it's a front-end only app calling a backend), AWS/Google Cloud, or even a Docker container hosted somewhere). The demo **must be live and usable** for at least the judging period, so make sure it's deployed in a stable manner. If you build a web UI, it should be intuitive: allow file uploads (multiple files) and have a text box for questions, with a display area for answers. If you go the API route, provide clear documentation and maybe a small script or collection for how to call it. The purpose of the live demo is to show that your solution actually works end-to-end in a real environment (not just on your local machine). This also makes it easier for judges to quickly test your system with custom documents and questions.
- **Functionality and Features:** Your solution should implement *all core features described in the Technical Specifications*. In particular, ensure that:
    - File upload and content ingestion works for all the formats listed (at least a basic level – for example, it's acceptable if certain advanced formatting in docs isn't perfectly preserved, but the main text content should be extracted).
    - The question-answering logic is in place and can handle queries across the ingested content.
    - The special edge cases are addressed: crawling external links (at least one level deep) from documents, performing OCR on images, using semantic search for retrieval, and handling structured data queries. We will test these aspects explicitly. For instance, we might provide a document that contains a link to a Wikipedia page and ask a question whose answer is only on that page – your system should ideally fetch that page and find the answer. Or we might give an image with text and ask about its content – your system should use OCR. Or a CSV with some numbers and ask an aggregate question. So be sure not to skip these features; even a basic implementation for each is better than ignoring it.
    - The system supports **English-language documents and questions**. We are not evaluating performance on other languages, so you can assume all input texts and user queries will be in English. (If you want to support other languages as a bonus, that's fine, but not required.)

- The user's question is answered in English clearly, and if the answer involves numeric or factual data, it should be accurate (for example, if the CSV says sales = 100, the answer shouldn't say 1000). When using an LLM, consider constraining it by providing context so it doesn't stray beyond the provided info.

- **Optional UI (or API):** A user interface is optional **but encouraged**. A simple web interface can greatly enhance the presentation of your project and make it easier to demo. However, if web development is not your forte, you can stick to exposing the functionality via an API. In either case, **the core requirement is a working Q&A functionality**. If you build a UI, it should at minimum allow the upload of documents and input of a question, and then display the answer. You might also display the source or highlighted text from which the answer was derived (this is a nice touch for transparency, but not strictly required). If building an API, ensure there are endpoints for uploading documents (or sending document content) and an endpoint for submitting a question and retrieving the answer. Document how to use these endpoints. We should be able to use your system without diving into the code – either through a friendly interface or well-defined API calls.
- **Testing and Generality:** A small test document corpus will be provided to you by the judges (e.g. a few sample files in the various formats with some known Q&A pairs) so you can validate your approach during development. *However, do not tailor your solution specifically to that corpus*. We will also evaluate your system on **hidden documents and queries** that you have not seen before. This is to test the generality and robustness of your approach. Make sure to test your system on a variety of documents beyond the provided samples – try different lengths, contents, and formats to see if it breaks or if there are bugs. The expectation is that your solution can handle arbitrary content reasonably well, rather than being overfitted to specific documents. The judges' hidden tests will include edge-case scenarios (like the ones described: an image-based document, a link that needs crawling, etc.) to ensure your system truly implemented the required features. To score well, your system should be able to successfully answer most questions on the hidden set, so focus on making your implementation robust and adaptable. If there are any limitations (for example, "our system struggles with very large PDFs of 1000 pages" or "the OCR might fail on handwritten text"), it's okay – document those and maybe handle them gracefully – but the more you can cover, the better.
- **Project Documentation:** Along with the code, ensure you provide adequate documentation and usage instructions. This includes:
  - A clear **README** in the GitHub repo.

- Comments in code where complex logic occurs.
- If you implemented a pipeline, perhaps a short description of your architecture (could be in the README or a separate design.md) explaining how you integrated the components (for example, “we first process the files with X library, then store texts in a vector index using Y, then use model Z to answer questions…”).
- Any **setup steps** for running locally, and how to deploy (if someone were to re-deploy the demo).
- List of team members (for credit) and any third-party resources used.

Good documentation not only helps the judges but also reflects professionalism and clarity of thought. We should be able to follow your instructions to run the system on our own with minimal fuss.

In summary, deliver a **Python-based Q&A system** that meets the functional specs, along with a GitHub repo of your code and a live demo link. Ensure it handles the diverse file types, uses AI/NLP to answer questions accurately, and covers the outlined edge cases. Only English needs to be supported. You can use external services for heavy-lifting tasks (OCR, embeddings, LLMs) as needed. We will test your solution on unseen docs, so make it general and robust. All these requirements aim to ensure that you build something that is not only cool and innovative but also **complete and reliable**. Good luck, and we can’t wait to see your creative solutions!

Tips and Best Practices

This project is certainly challenging – it involves combining file processing with advanced NLP. Here are some tips and best practices to guide you and help you make design decisions:

- **Plan Your Solution Pipeline:** Break the problem down into sub-tasks and design a pipeline. For example, one logical architecture is: **Document Ingestion** (file upload & parsing) → **Knowledge Indexing** (storing extracted content in a searchable form) → **Query Processing** (understand the question and search for answers) → **Answer Generation** (formulate answer from found info). Planning these stages will help you tackle the project step by step. You might implement and test each stage independently before hooking them together. For instance, first ensure you can extract text from all file types, then build the search/index component, then integrate

the Q&A logic. A modular approach will make debugging easier and your final system more maintainable.

- **Leverage Existing Libraries:** You **do not** need to code everything from scratch. Take advantage of libraries for parsing documents:
    - Use Python libraries like **PyMuPDF/pdfplumber or PyPDF2** for PDF text extraction (they handle PDFs reliably), **python-docx** for Word files, **python-pptx** (or Apache POI via JPype, or Aspose if available) for PowerPoint slides, and Python's built-in csv or pandas for CSVs. These tools will save you from dealing with low-level file format details and get you the text/content in just a few lines of code.
    - For images and scanned PDFs, integrate an OCR library. **Tesseract OCR** (via pytesseract in Python) is a popular open-source choice – just be sure to install the Tesseract engine and language data. If you need better accuracy and have internet access, you could consider an API like Google Cloud Vision or Microsoft OCR. Using such services can significantly improve text recognition quality for difficult images. Remember to only use OCR on images or PDFs that actually need it (you can detect if a PDF page has no extractable text and then fall back to OCR).
    - To implement semantic search, look into **embedding models** and **vector databases**. For example, **SentenceTransformers** (a Hugging Face library) provides easy-to-use embedding models for text. You can feed all your document paragraphs into the model to get vectors, and store them in a vector search library like **FAISS** or **Chroma DB**. This will let you quickly retrieve semantically similar passages for a query. Many teams also use **LangChain** – a framework that simplifies building such pipelines by providing components for document loaders, text splitters, embeddings, and LLM integrations. LangChain or similar frameworks can handle chunking documents into pieces and creating embeddings for you, which might speed up development. Don't reinvent the wheel for these standard operations; use these tools so you can focus on the overall integration and logic.
    - For Q&A and natural language understanding, you can use a pre-trained QA model (like bert-large-uncased-whole-word-masking-finetuned-squad for extractive QA) or simply use a large generative model (like GPT-3.5 or GPT-4 via API) that you provide context to. Each approach has pros and cons: a smaller QA model can run locally and gives you more control over ensuring the answer comes from the text (extractive QA will point to a span in the context), while a big LLM can generate more fluent answers and handle

summarization or reasoning but might hallucinate if not constrained. You might even combine them (use retrieval to get relevant text, then feed that to GPT-4 with a prompt like "Answer the question using only this text"). Choose whatever strategy you're comfortable with, given the time. Just make sure the model's answers are grounded in the document data.

  - When dealing with tabular data (CSV/Excel), using **pandas** to load the data can be very helpful. You can then either directly compute answers (if it's a straightforward numeric question) or even use a library like PandasAI or an LLM agent that can execute pandas code. Converting Excel to CSV (if needed) or using an existing parser (like openpyxl) will help extract data without manual CSV conversion.

- **Implement and Test Edge Cases Early:** Some of the trickiest parts of this project are the edge cases (links, images, etc.). It's a good idea to tackle those sooner rather than later:
  - For **link crawling**, decide on a method (for example, Python's requests to fetch the HTML and BeautifulSoup to extract visible text). Test it on a known URL to ensure you can get the content. Also, think about limiting scope: you likely should only fetch the exact URL referenced, and perhaps set a timeout to avoid waiting too long or getting stuck on slow sites. If a document might have multiple links, you could fetch all of them upfront during ingestion to include in the index.
  - For **OCR**, run a quick experiment with a sample image (you can create one or use an online example) to see that your OCR setup works and tune it if needed. Check if the OCR output needs cleaning (for instance, remove extra newlines or fix common errors). OCR can be slow on large images, so maybe convert images to grayscale or lower resolution if that speeds it up without losing accuracy.
  - For **semantic search**, verify that embedding-based retrieval is actually helping. You could do a quick comparison: ask a question where the wording is different from the document and see if the semantic search finds it whereas keyword search might not. Ensure your vector search returns texts with relevance – you might need to tweak the chunk size of documents (e.g., splitting documents into reasonably sized pieces for indexing: too large and they may contain unrelated info, too small and context might break). As a rule of thumb, splitting documents into chunks of a few hundred words that semantically make sense (like paragraph or section chunks) is effective.

    - For **structured data**, try some basic queries on your data structures. If using pandas, test a sample question like "max value in column X" by actually implementing a function or prompt that does it. Ensure that the system can distinguish when a question is referring to the data file versus the text documents (maybe based on keywords like "average", "calculate", or if the question explicitly names something from a table). You might integrate a simple rule-based approach: e.g., if the question contains certain words or you know the user uploaded a CSV and the question is about numbers, directly query the CSV.
- **Focus on Robustness:** A robust system is one that handles unexpected or tricky inputs without crashing and still provides useful output. Here are some robustness tips:
    - **Graceful error handling:** If a file fails to parse (corrupted PDF or an unsupported format), your system should not crash entirely. Handle exceptions during file processing by logging an error or notifying the user that a particular file couldn't be read, while still processing the others. Similarly, if the OCR fails to extract anything from an image, you might note "(No text found in image)" internally rather than breaking.
    - **Memory and performance:** Loading very large documents can be memory-intensive, and calling large models is time-consuming. To keep the system responsive, consider limits like: chunk very large text into smaller pieces and perhaps only keep embeddings (not the entire raw text) in memory if using a vector store. If using an API like OpenAI for each question, you might want to limit how much text you send in the prompt (hence retrieval of just relevant snippets is helpful). Also be mindful of the number of documents – since this is a hackathon, you can assume maybe on the order of tens of documents, not thousands. It's fine to optimize for the expected scale (e.g., a user might upload 5-10 files, not 10,000).
    - **Answer validation:** When an answer is produced by an AI, double-check if possible. For example, if using an extractive QA approach, you can have confidence the answer came from the text. If using a generative LLM, you could add prompt instructions like "If the answer is not in the text, say you don't know" to reduce hallucination. Another idea is to have the system return a snippet or source location for the answer – this not only increases trust but also helps you debug if the model is saying something not supported. Even though not strictly required, judges will be impressed if your answer comes with some evidence or at least clearly is derived from the docs.

    - **Single vs multiple document context:** Decide how you feed context to the Q&A model. If you retrieve multiple snippets from possibly different documents, you might concatenate them for the model to see. Ensure the concatenation is coherent or maybe prompt the model in a way to consider each piece. Too many context pieces might hit token limits of models like GPT, so you may need to only pick the top few relevant snippets. This is a design choice – just make sure the approach is consistent and justifiable.
- **User Experience Matters:** Even though this is primarily a backend/AI challenge, a good user experience can set your project apart. If you make a UI:
    - Keep it simple and clean. A single-page interface with an upload section and a question box is perfectly fine. Display the answer clearly, and perhaps list the filenames that were used or some indication of confidence.
    - You might allow multiple questions after uploading files (maintain the "session" of uploaded docs in memory so the user doesn't have to re-upload for each question). This makes it function like an interactive Q&A assistant over the corpus.
    - If time permits, a nice touch is to display the passage from which the answer was found, or highlight the answer in context. For example, after answering, you could show a sentence or two from the document as a quote. This isn't required, but it demonstrates transparency.
    - Make sure to handle edge input in the UI too – e.g., disable the Ask button if no documents are uploaded, or show a message if the user asks a question and no answer is found.
    - For an API, ensure you handle things like concurrency (if multiple requests come in) and have clear response formats (like JSON with fields for answer and maybe sources).
- **Testing and Iteration:** Use the provided sample corpus to test your end-to-end system, but also gather a few extra documents on your own to try out. For instance, you might take a textbook chapter PDF, a Wikipedia article saved as HTML (or PDF), a CSV of some public dataset, and an image with some text, and load them all, then challenge your system with questions. This can reveal integration bugs or performance issues. Encourage each team member to contribute test cases – sometimes a fresh set of eyes will ask a question you didn't think of. It's also a good idea to test what happens when the user asks a question that *isn't* answered in the docs – see that your system doesn't return nonsense. If it does, you might adjust it to return a "no answer found" message in such cases (perhaps based on a confidence threshold).

- **Creativity and Extra Mile:** Once you have the core functionality working, think about what creative elements you can add to stand out:
    - Perhaps an **intelligent query understanding** component: for example, if the user asks a very broad question, your system could respond with a brief summary from each document.
    - Or integrate a **feedback loop**: allow the user to mark if an answer was not helpful, and maybe log that for your own evaluation or adjust something.
    - Maybe a **visualization** for data answers: if a question asks for a trend and the data is in a CSV, you could even plot a quick graph as part of the answer. (This might be more than required, but it's the kind of feature that can impress if done right in a hackathon.)
    - **Caching**: If your demo will be used multiple times, you might cache embeddings or OCR results so that repeated uploads of the same file won't redo all the work. This is more of a optimization detail, but it shows thoughtfulness.
    - Ensure your solution is **secure** for a web deployment: this means be careful with file handling (no arbitrary code execution), and if using an API key, don't expose it on the client side.
- **Time Management:** Given the breadth of this project, prioritize core features first. A sensible order might be: get document upload & text extraction working for a couple formats (say PDF and DOCX) → integrate a basic QA (even if just keyword search or a simple embedding search + answer) to have an end-to-end pipeline → then add OCR for images → then handle other formats (PPTX, CSV, etc.) → then refine with semantic search, better answer formulation, and link crawling. This way, you always have a working baseline and you incrementally add the fancy features. If you run short on time, it's better to have all features implemented in a basic way (even if, say, semantic search is not ultra-optimized, or the OCR works but maybe not perfectly on all images) than to have a beautiful solution for PDFs but nothing for images or data. So aim for coverage of all requirements first, then improve each. Use your team's diverse skills: maybe one person focuses on the front-end or API, another on the document parsing, another on the QA logic. Work in parallel when possible and integrate frequently.

Finally, **keep the spirit of the hackathon in mind** – this is about learning, innovation, and collaboration. Don't be afraid to try novel approaches. If something doesn't work, adapt and iterate. The judges will highly value a solution that is **creative, robust, and handles the tricky cases gracefully**. We're excited to see how you integrate everything into a cohesive

question-answering system. Good luck, and have fun building your all-in-one document Q&A assistant! 🎉