System Design Document: Notes RAG Alchemist (AI Agent Prototype)

Overview and Assignment Goal

The Notes RAG Alchemist project, also known as the Planner RAG Compiler, automates the tedious task of cross-referencing lecture notes with reference textbooks. In a typical manual workflow, a student might read through handwritten or hastily typed lecture notes, then flip through textbooks to find relevant explanations or definitions. This system replaces that manual cross-referencing with an AI-driven pipeline. Given a lecture's notes (which may be scanned images or text PDFs), the system produces a comprehensive report that interweaves the lecture content with relevant textbook excerpts, complete with citations and figure references.

Assignment Goal: Build an AI agent prototype that can reason, plan, and execute the above task autonomously. The expected output for each lecture is a citation-rich, figure-aware and textbook-grounded PDF report. In particular, the system will:

- **Ingest Lecture Notes:** Accept a lecture note document (e.g. a scanned handwritten PDF) and segment its content[1].
- **Semantic Comparison:** For each segment, compare it semantically against a dataset of up to 10 reference textbooks (capped at ~1200 pages total)[1].
- **Relevant Excerpt Extraction:** Find highly relevant excerpts from those books that match the segment's content, and prepare accurate citations for each (including book title and page number)[1].
- **PDF Report Generation:** Assemble a PDF report containing:
    - Each lecture note segment (quoted verbatim from the input notes).
    - The matching reference book excerpts (quoted with citations in the format *Source: [Book Name], p.X* or *pp.X–Y* for page ranges)[1].
    - References to any figures mentioned (using figure identifiers along with source book and page)[1].
    - A brief trace of the planner's decision for that segment (e.g. "Match found" or a note that no match was found and an external reference is suggested)[1].

If a lecture segment has no good match in the provided books, the agent handles it gracefully. It will suggest an **External Reference** – e.g. name a standard textbook likely to cover that topic – and include that suggestion in the report (marked as an external source

recommendation)[1]. The user can then be prompted to add that book to the local dataset for future runs.

Architecture Diagram

Architecture diagram: The system's modules and data flow. Lecture notes are processed through OCR and segmentation, then each segment is routed through the Planner Agent and Retriever to query a Pinecone vector store of textbook embeddings. The Compiler then generates a PDF report combining the lecture and relevant excerpts.

The overall architecture follows a **Planner-Worker pattern**[1]. The system is not a single monolithic LLM agent; instead, it comprises a coordinating **Planner** agent and several specialized workers (*Retriever, Compiler*, etc.), orchestrated by a Streamlit-based UI. The diagram above illustrates the major components and their interactions, which are summarized here:

- **User Interface (UI – Streamlit app):** The user uploads input files via the UI. Lecture note PDFs flow into the OCR/Text Extraction module, while reference textbook PDFs flow into the indexing pipeline. The UI triggers each stage and displays status updates and results.

- **OCR & PDF Text Extraction (`pdf_utils.py`):** If the lecture PDF or a reference book PDF is scanned (non-selectable text), this module performs OCR to extract text. If the PDF already contains digital text, it uses direct extraction. The output is a set of text pages and segments.

- **Embedding Indexer (`embedding_indexer.py`):** This worker handles reference textbooks. It splits textbook text into manageable chunks, embeds each chunk into a vector, and stores them in the Pinecone vector database (including metadata like book name, page number, and any figure identifiers)[1]. This indexing step typically runs once per reference book (or whenever new books are added).

- **Planner Agent (`planner_agent.py`):** The planner is the core decision-maker. It takes each lecture note segment in sequence and decides how to retrieve relevant info. For each segment, it may perform one or more queries via the Retriever, possibly refine queries, or decide that no good match exists and mark the segment as external[2]. The Planner's logic is described in detail below.

- **Retriever Worker (`retriever.py`):** This component interfaces with the Pinecone vector store to perform semantic similarity search. It receives a query (usually the lecture segment text or a refined version) from the Planner and returns top-matching textbook excerpts with metadata[2]. The Retriever filters out low-relevance results (below a similarity score threshold) and structures each excerpt with fields like the excerpt text, source book title, page number(s), similarity score, and any figure numbers mentioned[2].

- **Compiler (PDF Generator – `compiler.py`):** The Compiler takes the Planner's decisions and the retrieved excerpts to generate the final PDF report. It formats each lecture segment followed by its matching excerpt(s), including citations and similarity scores[2]. It also prepares summary tables (planner decision trace) and appendices for figures and references as needed[1]. This module uses a PDF library to layout text, headings, and page numbers.

All these components work in a pipeline each time the user runs the tool for a given lecture. The UI orchestrates the flow: (1) reference books are indexed (if not already), (2) lecture notes are OCRed and segmented, (3) the Planner iterates over segments and consults the Retriever, and (4) the Compiler produces the output PDF.

Component Breakdown

Each component has a distinct responsibility in the system. Below is a breakdown of the major modules and how they coordinate:

Planner Agent: *Implements the logic for handling each lecture note segment.* It uses a threshold-based heuristic to decide if a retrieved excerpt is a good match. For a given segment, the Planner sends an initial search query (usually the raw segment text, possibly with minor tweaks like appending the word "figure" if the segment mentions a figure) to the Retriever[2]. It receives the top excerpts and checks the highest similarity score. If the top score exceeds an acceptance threshold (τ_accept, e.g. ~0.75–0.83), the Planner marks the segment as a match found (decision = "NORMAL", meaning no special action needed)[2]. If the score is below a lower threshold (τ_fail, e.g. ~0.45), the Planner considers the result insufficient and may invoke an LLM to help decide next steps[2]. With LLM assistance (Anthropic Claude or OpenAI GPT), the Planner can get one of three directives for the segment: NORMAL (accept no match), REFINE (get a refined search query), or EXTERNAL (no good match; suggest an external book)[2]. In a refine scenario, the Planner will re-query the Retriever with the new query and see if the score improves beyond a refinement threshold[2]. This adaptive loop can repeat for a limited number of attempts (e.g. max 6 refines) with gradually lowering thresholds to catch borderline cases[2]. If an external suggestion is returned, the Planner records the suggested textbook title for that segment[2]. Throughout this process, the Planner logs a short decision trace (e.g. "Refine & retry", "External book suggested", or reasons like "LLM: normal", "Lowered τ") which will later appear in the report to explain the agent's decision[2].

Arize's discussion of agentic RAG highlights an iterative refinement approach – if initial retrieval yields inadequate results, the agent can reformulate queries or try alternative strategies[1]. This aligns with our Planner's fallback behavior under weak or noisy retrieval conditions.

Retriever Worker: *Handles semantic search over the vector store of textbook content.* It uses OpenAI embeddings to convert the Planner's query into a vector and queries Pinecone for the top-*k* similar chunks[2]. The raw results from Pinecone include the stored metadata (book, page, etc.) and a similarity score. The Retriever then normalizes and filters

these results: it drops any matches below a minimum score threshold (e.g. 0.55 by default, to ensure quality) and removes near-duplicates[2]. It stitches together chunks that are from the same page and consecutive (to avoid breaking an excerpt mid-sentence if the textbook was chunked)[2]. Each result is formatted as an Excerpt object with fields:

- **text:** the actual excerpt text (a paragraph or two from the book).
- **book:** source book title.
- **page:** page number of the excerpt's start.
- **pages:** list of all page numbers covered by the excerpt (if it spans multiple pages, e.g. ["10","11"]).
- **score:** similarity score of this excerpt to the query.
- **figure_nums:** list of figure identifiers mentioned in the excerpt text (e.g. ["2.3(a)"])[2].
- **metadata:** any additional metadata (original chunk ID, etc.).

These structured excerpts are returned to the Planner[1][2]. The Retriever is also capable of providing a short text snippet summary of top matches (for instance, a few lines with scores and book titles) which the Planner can optionally include in its LLM prompt when deciding to refine or not[2]. After the Planner has made decisions for all segments, the Retriever can be invoked in a batch mode to fetch the final excerpts for each segment (skipping any marked external) via its `retrieve_for_segments` method[2].

Compiler Worker (PDF Generator): Using the fpdf2 library, the Compiler assembles the final PDF report from the Planner's output and retrieved excerpts[1][2]. It iterates through each lecture segment in order and writes:

- **Lecture Segment Content:** The text of the segment (as extracted from the notes) is printed first, to provide context.
- **Matching Excerpt(s):** One or more textbook excerpts that were retrieved as matches are inserted next. Each excerpt is quoted verbatim and followed by a citation in the format *(Source: Book Title, p.X)*[1]. If an excerpt spans multiple pages, the citation format uses a range (e.g., pp.10–11). The Compiler ensures excerpts are not excessively long in the report – if an excerpt's text exceeds a certain length (e.g. >4000 characters), it will truncate it for brevity[2].
- **Similarity Score:** For transparency, the similarity score of the top match is displayed (e.g., a line like "(score=0.81)"). This helps the user gauge how closely the excerpt matched the query.

In addition, the Compiler scans each segment and excerpt text for any figure references using a regex (e.g., patterns like *Figure 2.1* or *Fig. 2.3(a)*)[2]. All figure IDs encountered are collected. After processing all segments, the Compiler can insert two special sections:

- A **Planner Decisions Table** at the beginning, summarizing each segment, the Planner's decision, and its decision trace (for example, *Segment 3 – EXTERNAL –*

"No match, suggest Griffiths")[1]. This provides a quick overview of what the agent did for each part of the lecture.

- A **Figure Pointers Appendix** at the end of the PDF, listing each figure reference found and where to find that figure in the textbooks. For instance, an entry might read "Figure 2.3 – see Smith et al., p.45" if Figure 2.3 was mentioned and the excerpt came from page 45 of Smith et al.[1][2]. This feature helps the user locate diagrams or illustrations in the source books, since the report itself does not embed the images.

The Compiler is implemented to output either directly to a file (`compile_report()` saves a PDF to disk) or to memory (`compile_report_bytes()` returns the PDF bytes) for easy integration with the web UI download button[2]. This separation of formatting logic ensures the presentation layer (PDF generation) is decoupled from the retrieval and planning logic.

OCR Module: The OCR and PDF text extraction logic is part of `pdf_utils.py` (used both for lecture notes and any scanned textbooks). It first attempts to extract text using PyMuPDF (a PDF parsing library) which works for digital PDFs. If the text is too sparse or empty on a page (indicating a likely scanned image), it falls back to Tesseract OCR on that page[2]. To improve OCR quality, the module preprocesses images: it can deskew pages, apply thresholding and morphological operations to clean noise, and try different OCR segmentation modes (Tesseract PSM) depending on content layout[2]. For example, if a page seems to be a single uniform block of text (like a full-page paragraph), the system uses a page segmentation mode optimized for a single block; if that yields very little text (sometimes OCR misses multi-column layouts), it will retry with a fully automatic mode[2]. This two-pass strategy (uniform block first, then general mode if needed) is akin to a "Gemini" OCR approach to maximize text capture from varying layouts. The OCR module returns cleaned text for each page, and it also reports metadata per page – such as an ink ratio (percentage of dark pixels, to detect if a page likely contains text or is mostly whitespace) and a flag if the page text seemed noisy or unusually short[2]. The Planner and UI can use this to adjust behavior (e.g., if the entire lecture PDF has a high noise ratio, the Planner can bias toward more conservative decisions or more refinement attempts).

User Interface (Streamlit UI): The Streamlit app (`app.py`) provides an interactive front-end. It allows users to upload files and set a few options (like which LLM provider to use, or toggling OCR enhancements). Upon file upload, the UI:

1. **Loads environment keys** (for OpenAI, Pinecone, etc.) and validates that necessary API keys are present[2].
2. **Initializes or updates the Pinecone index** for reference books (this is cached so that re-running on the same books doesn't re-index unnecessarily)[2].
3. **Runs the PDF text extraction** on the lecture notes. A heuristic `is_sparse()` may be used to decide whether to force OCR on all pages; for safety, the current implementation often forces OCR for handwritten notes by default[2]. The result is a list of text pages and metadata (`OCRPageMeta` per page).

4. **Segments the combined lecture text** into chunks (segments) of a manageable size (around 3–4 paragraphs) using `pdf_utils.split_text`[2].
5. **Configures the Planner** (threshold parameters, which LLM model to use, etc.) by creating a `PlannerConfig` object[2]. Notably, it passes in a flag if the lecture OCR text seemed noisy – if a large fraction of pages were marked as low-quality OCR, it sets `noisy_bias=True` to encourage the planner to favor refine/external decisions[2].
6. **Iterates through each segment**, calling the Planner's `plan_segment()` with the Retriever. This happens sequentially, and the UI captures intermediate results to show a progress bar/status.
7. After planning, calls `retriever.retrieve_for_segments()` to fetch the final excerpts for each segment (skipping those marked external) and then invokes `compile_report_bytes()` to generate the final PDF in-memory[2].
8. **Displays the planner's decision table** and offers the PDF for download on the interface.

The UI is primarily there for user convenience and monitoring – it shows info like how many segments were identified, the similarity thresholds, and if the lecture OCR is suspected to be poor quality (in which case it warns that the agent might miss matches)[2]. The design keeps the UI layer thin; all heavy processing is in the workers or planner so that the core logic can be tested independently of Streamlit.

Data Design

The system passes around structured data objects (mostly Python dictionaries or dataclasses) to keep track of segments, queries, and retrieved content. Key data structures include:

- **Lecture Segment Plan:** For each segment of the lecture notes, the Planner produces a plan dictionary capturing its decision. Important fields in this `SegmentPlan` object include:
    - `segment_text`: the raw text of the lecture segment.
    - `decision`: what the Planner ultimately decided for this segment – typically "NORMAL" (a match was found or at least an attempt was made), "REFINE" (the query was refined and retried), "EXTERNAL" (no match, external source suggested), or "ERROR" (if something went wrong). In the final output, any non-"NORMAL" is effectively a special case to highlight.
    - `query_text`: the final query text used for retrieval. This might be the same as `segment_text` or a refined version (if the Planner rephrased it for a better search).
    - `suggested_book`: (only for EXTERNAL decisions) the title of the external textbook the agent suggests for this segment[2].
    - `best_score`: the highest similarity score returned by the Retriever for this segment's queries. If no excerpt was found, this may be `None`.

- `trace`: a short trace message explaining the reasoning (e.g. "Normal search", "External book suggested", "Refine & retry", "LLM: normal"). This is limited to a brief phrase (around 10 words max)[2].
- `excerpts`: a list of excerpt objects retrieved for this segment. This field is filled in after the Retriever fetches the actual text snippets (in the planning stage it may be empty until the final retrieval step)[2].
- `no_match`: a boolean flag (used internally) to mark if no local excerpt was ultimately used. For example, if `decision` is EXTERNAL or if the best score was below the threshold, `no_match` would be True.
- `noisy_bias`: a flag carried over from the lecture analysis indicating if the OCR was noisy; the planner records this mainly for transparency (it might append "(noisy)" in the trace to indicate it had low confidence due to OCR quality)[2].

These `SegmentPlan` dictionaries are the primary input to the Compiler when generating the report. They encapsulate everything needed: the original segment text, what to do with it, and any content to include.

- **Excerpt:** As described, each retrieved excerpt is a structured dictionary representing a candidate textbook passage. The fields of an excerpt include *text*, *book*, *page*, *pages*, *score*, and potentially *figure_nums* and *chunk_id*[2]. The citation formatting in the report is derived from the book and page/pages fields. For a single-page excerpt, the Compiler will format the citation as "(Source: Book Title, p.X)", whereas for multi-page excerpts it will use a range like "p.X–Y" or multiple page numbers as needed. This formatting logic was explicitly considered in the prompt design for the Compiler[1][2] to ensure consistency (e.g., using "p." vs "pp." correctly). All excerpt objects coming from the Retriever already include the necessary metadata, so the Compiler can just plug them into templates.

- **Embedding Metadata:** When reference book chunks are indexed into Pinecone, metadata is attached to each vector. This metadata typically includes:

  - `book`: the book title or filename.
  - `page`: the page number the chunk started on.
  - `pages`: if a chunk spans multiple pages (e.g., a paragraph started on one page and ended on the next), this might list both.
  - `text`: the raw text of that chunk (this is stored as well, so that Pinecone can return the text without requiring a second lookup).
  - `figure_nums`: any figure identifiers found in that chunk's text[2].
  - (Possibly other info like a `chunk_id`.)

Storing `figure_nums` in the vector metadata allows quick access to figure references. For example, when the Retriever returns a match, it already knows which figure numbers (if any) were mentioned in that excerpt. This is how the system can compile the **Figure**

Pointers appendix easily – by aggregating figure IDs from all excerpts. Moreover, by including page numbers in metadata, the agent ensures that citations can be precise. The vector store doesn't store entire PDFs or images – only these text chunks and their tags – which keeps it efficient and means all content that goes into the final report comes from the user-provided books (there's no retrieval of unseen content).

- **Citation and Figure Reference Handling:** The design of data structures ensures that every excerpt knows its origin. The Compiler uses simple helper functions to format page ranges and figure listings. For instance, if an excerpt's metadata has `"pages": ["10","11"]`, it knows to format the citation as *pp.10–11*. If an excerpt (or the lecture segment text itself) contains the substring "Fig. 2.3(a)" or "Figure 2.3", a regex will capture "2.3" as a figure ID and store a tuple like `("2.3", "Book Name", (2,3), 2)` in a set of figure references[2]. Here, 2.3 is the figure label, *Book Name* comes from the excerpt's book field, (2,3) might be the tuple of pages involved, and 2 an integer of the first page for sorting. This data design makes it straightforward at the end to iterate over the unique figure references and produce a sorted list in the appendix (grouping by book and figure number)[2].

Overall, the system favors simple Python native structures (lists of dicts, etc.) for portability. This also made it easier to write unit tests by constructing fake segment plans or fake retrieval results as dictionaries, without needing complex object serialization.

Technologies Used and Rationale

The project employs several technologies and libraries, each chosen for a specific purpose in the architecture:

- **OpenAI and Anthropic LLMs:** The system uses OpenAI's models for two main tasks: obtaining text embeddings and (optionally) aiding the Planner's decisions. For embeddings, the OpenAI `text-embedding-3-large` model (with 3072-dimensional vectors) encodes text chunks and queries[1]. This model was chosen for its strong semantic capabilities on lengthy texts. The Planner's refine/external logic can use an LLM; the implementation supports OpenAI and Anthropic Claude as providers[1]. Claude was experimented with for natural language planning tasks due to its lengthy context and reliable output formatting, whereas OpenAI's GPT was used for the workers and could also be used for the Planner if Anthropic is not available. I selected Anthropic Claude for the planner because its behavior supports an iterative search/refinement loop (i.e., issuing an initial query, inspecting results, then refining subsequent queries to improve retrieval quality). These capabilities are described in Arize's "Understanding Agentic RAG"[5]. By relying on hosted LLM APIs, the project avoided maintaining any local model infrastructure, keeping the prototype lightweight.

- **Vector Database (Pinecone):** Pinecone's cloud vector store is used to index and query the textbook embeddings[1]. Pinecone was selected for its ease of use and

performance on similarity search, especially given the free tier can handle the project's scale (~1200 pages across books) comfortably. It abstracts away the complexities of ANN (Approximate Nearest Neighbor) search and scaling. In an extensibility note, the code is written such that swapping in a local vector DB (like ChromaDB or FAISS) is feasible if needed[1] – for instance, `vector_store.py` has a layer of abstraction where all Pinecone-specific calls go through, making it easier to redirect to another backend. But for the prototype, Pinecone gave a quick, reliable solution.

- **OCR: PyMuPDF & Tesseract:** For reading PDFs, the combination of PyMuPDF (also known as *fitz*) and Tesseract OCR was used[1]. PyMuPDF is very fast at extracting text from PDFs that have embedded text (like generated PDFs or PDFs saved from Word). Tesseract is an open-source OCR engine good for scanned images. The system leverages both: first trying PyMuPDF for any extractable text, then falling back to Tesseract for image-based pages[2]. This dual approach ("scan if needed") provides robustness. Other OCR options considered include external OCR APIs (like Google's or Azure's) or more advanced models, but Tesseract was chosen due to its offline availability and sufficient accuracy for clear lecture scans. Basic image preprocessing (via OpenCV and PIL) was added to boost Tesseract's accuracy without requiring heavy deep-learning OCR models.

- **PDF Generation: fpdf2:** The report is generated with **fpdf2** (a Python PDF library)[1]. fpdf2 was chosen for its simplicity and control – it allows precise placement of text, custom fonts, etc., without needing a full TeX/LaTeX installation or heavy dependencies. It's a programmatic way to draw each page, which fits the need to dynamically create sections like the Planner's table or figure appendix. Alternatives like ReportLab or building HTML+CSS to convert to PDF were considered, but fpdf2 provided a good balance of low-level control and ease of development.

- **Streamlit (UI framework):** Streamlit was used to quickly create a web-based interface for the prototype[1]. I chose Streamlit to build the UI because it let me ship a working prototype quickly and stay entirely within Python. Given the 3–4 day timeline and the fact that I'm not yet strong in separate frontend (React/Next.js) and backend (Flask/FastAPI) stacks, Streamlit minimized glue code and still provided a decent UX (file uploads, buttons, progress, simple tables). Streamlit allows a Python script to become an interactive app with minimal overhead; it handles file uploads, buttons, and display elements in a simple manner. The result is a local web app where the user can upload notes and books and get a downloadable report. Since the target environment is likely a local machine (no persistent server), Streamlit was ideal – it runs locally and even the Pinecone and LLM calls are made from the user's session. In a more production setting, a React or Flask app could be considered, but for a prototype, Streamlit maximized development speed and provided a decent UX (e.g., progress messages, dataframes for the plan table preview, etc.).

- **Testing: Pytest:** The project uses **Pytest** for testing various components[1]. Unit tests cover functionality like the Retriever (with a fake in-memory vector search to ensure filtering and stitching logic works), the Planner's decision logic (by mocking LLM responses to simulate different scenarios), and the Compiler's formatting (by inspecting output for expected sections). Pytest was chosen for its simplicity and widespread use in Python projects. The tests helped ensure that, for example, low-score excerpts are filtered out, figure regex captures the intended patterns, and the Planner's state machine behaves as expected under different conditions. Given that the system integrates with external APIs, tests were designed to use stubs/mocks where possible (to avoid calling OpenAI or Pinecone in every test run). This testing approach ensures each module can be validated in isolation, which is important for catching regressions in a multi-component system.

In summary, the tech stack was selected to meet the prototype's needs with minimal friction: relying on proven cloud services (OpenAI, Pinecone) where appropriate, and using accessible open-source libraries (Tesseract, fpdf2, Streamlit) to handle the rest. This allowed focusing on the design of the solution rather than reinventing foundational tools.

OCR and Input Handling

Handling input robustness was a crucial part of the design because lecture notes can be messy. The system had to support both scanned handwritten notes and digital PDFs, and similarly, reference textbooks could be PDFs with embedded text or scans of old books. The approach to OCR and text extraction is as follows:

- **Direct Text Extraction vs OCR:** For each PDF, the system first attempts direct extraction. Using PyMuPDF, it can extract text from any page that has a text layer (common for PDFs generated from Word, LaTeX, etc.). If PyMuPDF returns sufficiently large text for each page, we assume the PDF is digital. However, if a page comes back essentially empty or with very few characters, it likely means the page is an image scan. The function `is_sparse(pages)` computes the total characters per page and compares against a threshold[2]. If the average text density is very low (e.g., each page has only a few words or none), the UI will trigger OCR mode for the entire document. In practice, for lecture notes, the app defaults `force_ocr=True` since they are often handwritten or have diagrams[2]. For reference books, it might do a page-by-page check: any page that failed text extraction is passed to OCR.

- **OCR Preprocessing:** When OCR is needed, the system uses pdf2image to render PDF pages to images and then processes them. Preprocessing steps include converting to grayscale, applying deskew (to straighten rotated text)[2], and using morphological filters to clean noise (e.g., dilating then eroding to connect broken characters)[2][2]. These steps improve Tesseract's accuracy on scans with faint or skewed text. The OCR is done via pytesseract with a configurable page segmentation mode (PSM). By default, a fully automatic page layout mode is used,

but if the user or system suspects uniform text blocks, a different PSM (like "single column" or "single block") can be tried. The code actually tries PSM 6 (assuming one block of text) and if the result looks too sparse, it falls back to PSM 3 (automatic mode) on the same image[2][2]. This dual attempt is what the prompt referred to as Gemini OCR – effectively two passes to catch text in tricky layouts (e.g., first pass might miss multi-column text, second pass catches it).

- **Handling of Mathematical or Unusual Notation:** The lecture notes may contain mathematical symbols or shorthand that OCR might garble. The system's text normalization (`normalize_ocr_text`) removes obvious OCR artifacts (like isolated @ symbols or trademark signs that sometimes appear)[2]. It also joins hyphenated line breaks (common when OCR splits words at end of lines)[2] and standardizes quotes and dashes. This doesn't solve complex OCR errors, but it cleans up the text so that the embeddings and LLM aren't thrown off by odd characters. For very math-heavy notes, an improvement could be to integrate a math OCR or leave LaTeX in place, but that was outside current scope – the assumption is that most content is narrative or definitional text.

- **Input Formats:** The system is built primarily for PDF inputs (for both notes and books). If needed, it could be extended to images (by converting images to a PDF or directly feeding them to the OCR module). The output report is always a PDF. The choice of PDF for output is because it's self-contained (text and formatting fixed) – ideal for a report that needs to preserve layout across different readers.

- **Heuristics for Mixed Content:** Some reference PDFs might have a mix of scanned pages and digital text (for example, an old textbook scanned but with a digital overlay for text). In such cases, PyMuPDF might extract some gibberish text (hidden text layer) or nothing. The design leans on the side of performing OCR if there's any doubt. It also logs per page if OCR was used (`used_ocr=True` in the metadata) so one can inspect later how many pages required OCR[2]. (The assignment brief mentioned an optional "Gemini" OCR API integration; in our implementation, no external OCR API is directly called – "Gemini" here refers to the twin-pass Tesseract strategy.) If needed, plugging in an API like Google Vision OCR or Azure OCR could be done in `pdf_utils.py` for potentially better accuracy on handwriting or complex documents, but that introduces external dependencies and cost.

- **Multilingual Notes (Not currently handled):** The project assumes English-language content[1]. If notes or textbooks were in other languages, one would need to use Tesseract language packs or a different embedding model that supports multilingual semantic search. Since the focus was a controlled prototype, we didn't integrate these; but the modular design (especially if using a local vector DB and adding language detection) could allow extending to other languages in the future.

In short, the system is designed to be resilient to imperfect inputs through OCR and cleaning. It uses simple metrics (character count heuristics) to decide when to OCR,

ensuring that scanned lecture notes don't slip through without being read. The result is that whether the user uploads a clear text PDF or a photo of handwritten notes/scanned textbooks, the backend will extract as much text as possible for the subsequent retrieval process.

Extensibility and Testing Hooks

While this is a prototype, it was built with extensibility and testing in mind. Several design choices make it easier to extend or adjust the system for future needs:

- **Pluggable Vector Stores:** The retrieval layer is abstracted such that one could replace Pinecone with a local store. For example, the `vector_store.py` module defines `search_vector_store()` and `init_vector_store()` generically. The code already hints at ChromaDB as an alternative[1]. To use a local vector DB, one would implement the same interface (initialize index, upsert embeddings, query by vector) in that module. The Planner and Retriever wouldn't need any changes – only the underlying store implementation would change. This is valuable if, say, the project needs to run entirely offline (no Pinecone) or if scaling beyond Pinecone's free tier limits.

- **Support for Different LLMs or No LLM:** The Planner was designed to not strictly require the LLM step. It has a configuration flag `llm_enabled` which can be turned off, in which case it will skip all refine/external suggestions and just use thresholds[2][1]. This is useful for testing or if API quotas are an issue – the system will still function in a degraded mode (it will rely purely on the similarity scores). Additionally, adding support for another LLM provider would just mean writing a new `_call_llm_planner_x()` function and adding it to the `PROVIDERS` map in `planner_agent.py`[2]. The prompts themselves are externalized in a `prompts/` directory (the Planner prompt template is read from a file, making it easier to tweak without changing code)[2].

- **Testing Hooks:** The codebase includes unit tests (not shown here, but described in the design). For instance, a dummy retriever class can be injected to simulate search results, allowing testing of the Planner logic deterministically[2]. The Planner's functions are structured to be stateless given inputs (except logging), so they can be called in tests with mock data. The Compiler was tested with a "smoke test" where a small set of fake segment plans (with known excerpts and figure mentions) is passed to ensure the PDF creation doesn't error and output contains expected text[2]. Also, the presence of fields like `best_score` and `trace` in the plan makes it easy to assert certain conditions (e.g., if `decision` is "EXTERNAL", then `best_score` should be below threshold and `suggested_book` not null, etc.). The project documentation mentions unit and integration tests for the Retriever (with and without Pinecone), as well as tests for the Planner with mocked LLM responses[1]. These ensure that:

    - The Retriever correctly merges and filters results (e.g., no duplicate excerpts, and multi-page excerpts are stitched).
    - The Planner's loop respects the max steps and thresholds (e.g., it doesn't run away in infinite refine, and it correctly identifies when to switch to EXTERNAL).
    - The figure regex catches variations like "Fig 2.1" vs "Figure 2.1(a)"[2].

- **Logging and Debugging:** All key decisions are logged (the Planner writes a log file of all segments' decisions with hashes for segments)[2]. This is useful for debugging on large inputs – one can see after the fact which segments failed or triggered external suggestions. Additionally, the Streamlit UI's dataframes (for the planner records and index log) act as a form of lightweight monitoring, so a user can spot if many segments got no match or if OCR yielded 0 chunks from a reference (which likely means OCR didn't work on that PDF).

Extending to New Features: The design is modular enough to consider some extensions with minimal changes:

- *Multilingual support:* As noted, adding multilingual embedding models or OCR would be a matter of injecting those at the right place (e.g., use a language detector on the notes, choose an embedding model accordingly, or run Tesseract with a different language for non-English text).

- *Additional modalities:* If one wanted to include figure images in the report, the system could be extended by storing figure images when indexing (e.g. detecting figure numbers and caching the figure picture). The figure appendix could then embed those images. This would mainly involve extending the indexer to capture images from PDFs (using PDF rendering or if images are separate), and extending the Compiler to place images. The current design did not do this (for simplicity, figures are just referenced by ID and source page, not extracted)[1].

- *Hosted Deployment:* To deploy this as a service, the Streamlit UI would be replaced or supplemented with an API endpoint. The core logic (planning, retrieving, compiling) could remain the same. Because state is mostly passed through function calls, it wouldn't be hard to wrap it in a Flask or FastAPI endpoint that accepts files and returns a PDF. One would just need to manage the Pinecone index persistence (currently it expects to run continuously with an in-memory cached index; a production version might initialize the index on each run or use a persistent volume).

- *Scaling dataset:* The design decision to limit to ~10 books / 1200 pages[1] was for the prototype (to stay within free API limits). If scaling to more books, one might implement a more sophisticated retrieval that first narrows down which book to search (e.g. clustering by book) or adds an intermediate step of keyword retrieval. The current structure can handle moderately more data but beyond a point,

Pinecone queries might need namespace partitioning or metadata filters by subject. These can be integrated by extending the Retriever's query to include filtering by book or section if hints are known (the code already passes a namespace parameter around, which could be used to segregate indexes by topic or course if needed)[1].

In summary, the system was built not just to solve the immediate task, but as a flexible framework that can be adapted. The combination of configuration flags, isolated modules, and thorough testing means a future developer or researcher can tweak components (swap models, add new output sections, etc.) with confidence that the rest of the system will continue to work.

Optional Improvements

Finally, during development a few improvements and refinements were noted as "nice-to-have" if time permitted. These did not make it into the initial prototype but are worth mentioning as they indicate areas for polish:

- **Table of Contents (ToC) Realignment:** The report currently generates a Planner Decisions table and then immediately the content segments. A more polished report would include a proper Table of Contents page that lists each segment (perhaps by a short title or topic) and the page number on which it appears in the PDF. The infrastructure for this is partly in place – the Compiler was tracking `toc_entries` and even calculating the ToC after writing pages[2]. However, some layout issues remained (ensuring the ToC itself doesn't push content to new pages, aligning dot leaders, etc.). Fine-tuning this would enhance readability for long reports by letting the reader jump to a particular segment of interest.

- **Better Segment Titling:** As seen in the sample output[3], the segment titles are auto-generated by taking a few keywords from the segment text. This sometimes yields awkward phrases (e.g. fragment words or all lowercase). Improving this could involve using an LLM to summarize each segment into a title, or at least refining the heuristic to avoid uncommon terms or include key nouns. Even simply using the first sentence of the segment as a title (truncated to a reasonable length) could be more understandable. This would reflect in both the ToC and as a heading before each segment's content in the report, making the document look more like a structured report rather than just a raw dump.

- **Clearer Fallback Messages:** When no match is found for a segment, the current report might just show the segment and perhaps an empty excerpt section or a note like "(No excerpt found)". It would be better to explicitly state something in the report like: "No relevant excerpt found in provided references. Suggested reading: *[Book Name]*." This way, a reader of the PDF (who might not have the UI context) understands that the agent didn't find anything and recommends a certain

textbook. Implementing this would be straightforward by checking the `no_match` flag in the Compiler and inserting a line for external suggestions.

- **Layout and Aesthetics Refinements:** The PDF layout can be further improved. Potential changes include:
    - Using different font styles or sizes for lecture segments vs excerpts to differentiate them (currently, all text appears similar; using italics or indentation for excerpts could help).
    - Adding section numbering or decorative elements for each segment section.
    - Ensuring that a long excerpt or segment doesn't split awkwardly across pages (the Compiler does some guarding, but one could force a page break before a segment if not enough space).
    - Including metadata like the course name, lecture date, or a header/footer on each page. For example, a footer could say "Lecture Notes Report – Page X" for professionalism.

- **Performance Optimizations:** For very large inputs, some optimizations could be added. Caching of OCR results (so re-running on the same lecture doesn't OCR again) or caching Pinecone queries for repeated segments (not typical, but if a segment's text repeats, you could reuse results) are ideas. Also, parallelizing the retrieval for each segment could speed up processing (currently the Planner loop is sequential due to its logic, but segments are independent once queries are decided).

- **Multithreading or Async Agents:** As an optional feature mentioned in the assignment, multi-agent collaboration was considered. In this prototype, the Planner and Retriever act more like a synchronous pipeline rather than fully independent agents. An improvement could be to run the Retriever as an asynchronous service or in a separate thread – the Planner could send multiple queries in parallel if segments were independent. However, since the Planner's logic can depend on iterative refinement, true parallelism is non-trivial. A compromise would be at least parallelizing the embedding of lecture segments or performing OCR on pages in parallel. (Streamlit does not easily allow multi-threading in the same script, so this would have to be handled on the back-end side.)

- **API Integration & Scheduling:** As a forward-looking idea, one could wrap this system into a scheduled job (for example, automatically generate reports every week for new lecture notes) or allow it to pull reference materials from an online source if not provided (e.g., automatically download a public textbook PDF when suggesting an external reference – with proper permissions). This would make the agent more autonomous. Currently, it relies on the user to provide all data.

- **Scalability & Deployment:** The current Streamlit app runs in a single-user session and does not scale to multiple concurrent users; there's no dedicated API layer, queueing mechanism, or persistent service. To move toward production, one would migrate to a front-end + back-end architecture (e.g., React/Next.js for the front-end and FastAPI/Flask for the back-end), add an asynchronous worker/queue system (Celery or RQ) for long-running tasks, implement persistent storage for indices and logs, add authentication and rate-limiting, and use containerization (Docker) with a small vector database service. This would enable multi-user access, better observability, and controlled throughput while keeping the RAG core unchanged.

Each of these improvements targets a more polished and robust user experience. They were not strictly required to demonstrate the core functionality (which is already achieved – given notes and books, produce a matching report), but addressing them would move the prototype closer to a production-ready educational tool. In a future development context, these are natural next steps to discuss and plan after the initial successful prototype.

References:

- [1] Full Project Summary for Lecture Notes AI Agent Prototype (assignment brief and system description).
- [2] Notes RAG Alchemist Code Repository (GitHub, commit 3af6d6f, includes Planner, Retriever, Compiler modules).
- [3] LectureAssist Example Output Report (sample generated report used to infer formatting and improvement areas).
- [4] OpenAI ChatGPT (GPT-4 model, conversation output used for initial content drafting).
- [5] LaViale, Trevor. “Understanding Agentic RAG.” *Arize AI Blog*, Feb. 5, 2025. [Online]. Available: https://arize.com/blog/understanding-agentic-rag/

---

[1] Understanding Agentic RAG - Arize AI

https://arize.com/blog/understanding-agentic-rag/