ChatGPT

Designing a Retrieval-Augmented Generation (RAG) Knowledge Base System

Overview and Objectives

This report presents a comprehensive design for a Retrieval-Augmented Generation (RAG) knowledge base system that is **scalable** and **cost-efficient** (targeting under \$100/month). The solution leverages GitHub for code/integration, Azure for cloud services (including Azure Active Directory for user authentication), and Vercel for hosting a Next.js front-end. Initially, FERPA compliance is not required, but the design will allow future enhancements for data privacy. The core idea is to build a system where documentation data is automatically ingested into a vector-indexed knowledge base, and a chatbot interface uses that data to answer users' questions with accurate, context-backed responses. The design considerations include automation of data updates, an interactive chat UI, text chunking and embeddings, data integrity, and performance optimizations.

1. Automated Documentation Pipeline (Python)

A robust Python-based pipeline will periodically crawl documentation sources and update the knowledge base. Key recommendations include:

- **Crawling Tools:** Use a dedicated web scraping framework like **Scrapy** for efficiency and scale. Scrapy supports asynchronous crawling and has built-in handling for requests, sessions, and data pipelines [1] [2]. Its asynchronous design allows crawling many pages concurrently, which is useful if documentation sites are large. For simpler scenarios or quick prototypes, **BeautifulSoup** (with the `requests` library) is an alternative for parsing HTML after fetching pages [3]. BeautifulSoup is easy to use for small-to-medium sites and offers fine-grained parsing control [3] [4], but it lacks Scrapy's built-in scheduling and asynchronous throughput. In practice, Scrapy can form the backbone for scalable crawling, while BeautifulSoup might be used for specific parsing tasks within Scrapy callbacks. The pipeline should be designed to accommodate both public docs and, in the future, **authenticated internal sources** (e.g. behind SSO or on private repos). This means structuring the code to allow plugging in different data sources – for instance, using Scrapy's ability to set custom request headers or session cookies for authenticated pages, or using APIs/SDKs for sources like Confluence or GitHub (for private markdown files).

- **Incremental Updates:** To avoid reprocessing unchanged content each week, implement an **incremental crawling** strategy. One approach is to track document signatures (timestamps or hashes) and only process new or updated pages. For example, Scrapy's `deltafetch` plugin can skip previously seen pages based on a cache of request fingerprints [5]. You might periodically crawl top-level index pages or sitemaps to discover new links (as is common on news or docs sites) [6]. Then maintain a record (in a lightweight database or file) of which page URLs have already been ingested. On each run, compare the newly found links with those already in the knowledge base [7].

For pages that were seen before, conditional HTTP requests (using `ETag` or `Last-Modified` headers) can check if content changed; if not, skip re-downloading to save bandwidth. This incremental approach reduces redundant processing by **focusing only on new or changed documentation** [8] [9]. It's also wise to modularize the crawler per documentation source – e.g., one spider for public docs, another for an internal repo – so that each can be updated or run independently.

- **Scheduling and Automation:** Schedule the crawl and update pipeline to run **weekly** (or at whatever interval balances freshness with cost). For simple, budget-friendly scheduling, a cron job on a server or a GitHub Action triggered on a schedule can suffice. A cron job is straightforward but lacks monitoring and retry – if a run fails, cron won't automatically retry or alert. For more robustness, consider **Apache Airflow** or **Celery Beat** if the pipeline grows complex or requires coordination. Airflow is a workflow scheduler that allows building DAGs (task pipelines) with features like retries, logging, and dependency management [10] [11]. It excels at orchestrating multi-step jobs (e.g., crawl -> process -> index) and handling failures (with retry policies) [10]. However, Airflow's overhead (maintaining a scheduler service and database) might be overkill for a single weekly job and could strain the \$100 budget. **Celery** with a message broker (and Celery Beat for periodic tasks) is another option: it allows distributed task execution and scheduling within a Python app [12]. Celery would let you queue the scraping task and any post-processing asynchronously, but it requires running a worker (and a broker like Redis). For cost and simplicity, a good strategy is to start with a **cron or scheduled GitHub Action**. For example, a GitHub Action can run a Python script weekly to crawl docs and then push updated data (such as new embeddings) to your database or storage. This leverages free CI minutes (within reason) and keeps costs low. As needs grow, you can "upgrade" to Airflow for better monitoring (Airflow provides a UI to see task status and logs, plus easier DAG management) [13] [11].

- **Efficiency and Cost:** To keep the pipeline affordable, optimize what data is processed and stored. Implement **incremental text embedding** updates: if a document hasn't changed, avoid re-embedding its text (embeddings can be computed once and reused). This can be achieved by storing a hash or checksum of each document's content alongside its embeddings. On each weekly run, before processing a doc, compute its current hash and compare to the stored hash; only re-embed if different. This ensures you don't pay for embedding the same text repeatedly. Additionally, control the crawl rate and scope to avoid hitting site rate limits or downloading huge amounts of data unnecessarily. Focus on the sections of documentation relevant to your domain (you might skip auto-generated API references, etc., if not needed). If using cloud functions or jobs for crawling, ensure each run stays within free tier limits if possible (Azure Functions, for example, could run a Python crawler on a timer trigger; if execution is under a certain time, it might stay in the free quota). Logging is important for maintainability – have the pipeline output what it did (pages fetched, new content found) to a log that can be reviewed, especially as you add internal sources later (which may require debugging auth issues).

- **Future Integration (Internal Repos):** When incorporating private documentation (for example, a private GitHub wiki or an internal markdown repository), use appropriate APIs rather than web crawling where possible. For instance, the GitHub API can list files in a repo and retrieve raw content. This is often more reliable than scraping HTML. You could write a Python module that authenticates with a GitHub Personal Access Token and pulls markdown files from the repo, then feeds them into the same chunking/embedding process. Similarly, for an internal site requiring Azure AD auth, you might use an automation account with credentials or tokens to fetch data. **Design the pipeline as a collection of plugins or sources** – e.g., one source is "Public Docs via crawl", another is "Internal Markdown via GitHub API", etc. This modularity makes the system extensible. Security-wise, store any credentials (like API tokens) securely (in Azure Key Vault or GitHub Actions secrets, etc.) and ensure they aren't exposed in logs. Although FERPA isn't a concern initially, it's prudent to handle any credentials or data access with best practices (encrypted storage, minimal permissions) so that later compliance needs (ensuring student data privacy) can be met more easily.

Potential Challenges & Solutions: One challenge is that documentation websites can change structure or require JavaScript. If you encounter client-side rendered docs (SPA), a headless browser or alternative approach might be needed (e.g., use **Playwright or Selenium** to render pages, though these are heavier). This would increase complexity and cost, so prefer sources with static HTML if possible. Another challenge is ensuring data quality – the pipeline should strip irrelevant parts (navigation menus, footers) and possibly segment content by sections, so the knowledge base isn't polluted with irrelevant text. Using Scrapy's parsing or BeautifulSoup, you can target specific HTML containers (for example, the main content div of docs) to extract clean text. Implement robust error handling: if a particular site or page fails, catch exceptions so that one failure doesn't abort the whole pipeline. Logging and alerting (e.g., email on failure) will help maintain this long-term with minimal manual effort. Overall, the automation pipeline should run quietly in the background, with cost control via incremental updates and careful choice of tools (Scrapy for big jobs, simple scripts for small ones) to meet the budget.

2. Interactive Chatbot Interface (TypeScript/Next.js)

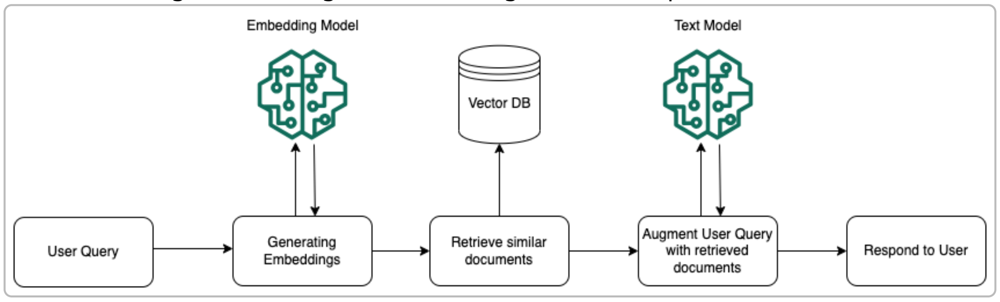
The user interface will be a web chatbot that allows users to ask questions and receive answers augmented with documentation context. This will be built with **TypeScript and Next.js**, following modern best practices for a responsive, secure application:

- **Frameworks and SDKs:** Next.js is ideal for this project due to its ability to handle server-side API routes and static front-end pages in one framework. We recommend leveraging the **Vercel AI SDK** (a toolkit for building AI applications in TypeScript/React) to simplify the chat implementation. The Vercel AI SDK provides React hooks and utilities for streaming responses from language models and managing chat state. For example, it offers a `useChat` hook that can handle messaging and stream tokens from the server to the UI in real-time [14]. This creates a smooth, token-by-token streaming chat experience for the user. Vercel's SDK is also designed to integrate retrieval-augmented generation patterns out-of-the-box – their templates demonstrate how to call a vector database and feed results into the model prompt [15] [14]. Alongside the Vercel SDK, consider the **OpenAI Node SDK** (OpenAI's official library for Node.js) for making API calls to Azure OpenAI or OpenAI endpoints. The OpenAI SDK will be used server-side (in Next.js API routes) to generate embeddings (for queries) and to get chat completions. It's straightforward to use and supports streaming responses. **LangChain** is another option – it provides higher-level abstractions to implement a QA chain (retrieval + LLM) and is available in JavaScript/TypeScript (LangChain.js) as well as Python. If the logic of retrieving documents and formatting the prompt becomes complex, LangChain can help manage it. However, adding LangChain also adds overhead; for a relatively simple QA bot, you might implement the logic directly using the OpenAI SDK and a vector search call. In summary, the front-end will use **Next.js with React** (possibly with Tailwind CSS or Chakra for styling) and rely on either direct API calls or the Vercel AI SDK for the chat functionality.

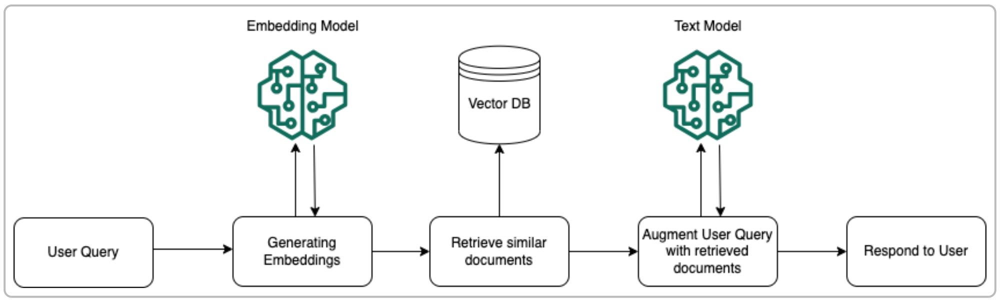
- **Chatbot Retrieval Workflow:** The chatbot will operate on the classic RAG workflow – when a user asks a question, the system will retrieve relevant context from the knowledge base and use it to formulate an answer [16]. Specifically, the Next.js API route (e.g., `/api/chat`) will handle incoming queries by performing the following steps:

- **Question Embedding:** The user's query is embedded into a vector using the same embedding model used for the documents. If using Azure OpenAI or OpenAI, the query can be sent to the embedding endpoint (e.g., text-embedding-ada-002). This results in a numeric vector representing the query's semantics.
- **Vector Search:** Query the vector database or index for similar vectors, to retrieve the top *k* most relevant document chunks. This uses cosine similarity or dot-product to find which document chunks are semantically closest to the question. The result is a set of text snippets (with their source metadata) that are likely to contain information to answer the question.
- **Augmenting the Prompt:** Construct a prompt for the language model that includes the user's question and the retrieved context. A common prompt template might be: *"You are an assistant answering questions with provided documentation. Answer the question using only the information below. If the answer is not in the documentation, say you don't know. Documentation: [snippet1] [snippet2] … Question: [user's question]"*. By prepending the relevant documentation text, we ground the LLM's answer in real data [16], reducing hallucinations.
- **Generating Answer:** Call the OpenAI/Azure OpenAI completion API (e.g., GPT-3.5 Turbo) with the augmented prompt. The model will then produce an answer that hopefully cites or uses the provided context. Using the streaming capability (either via the SDK or by handling the event stream from OpenAI), we stream the answer tokens back to the frontend for a live typing effect [14].
- **Post-processing:** Optionally, format the answer to include citations. For example, if our context snippets have source IDs, we can prompt the model to output answers with references like "[1]" corresponding to a source snippet. Alternatively, after receiving the model's answer, we could programmatically append a list of sources used (since we know which snippets we retrieved). The UI should display these citations as footnotes or clickable links for transparency.

Diagram – RAG Query Flow: To illustrate the runtime retrieval process, below is a schematic of how a user query is processed through embedding, retrieval, and generation steps



. The **user's question** is converted to a vector (via the embedding model), the **vector DB** is queried to retrieve similar document chunks, and those chunks are used to **augment the prompt** to the text generation model, which then **responds to the user** with an answer grounded in the docs. This is the core loop of the chatbot's back-end.



RAG runtime flow: the query is embedded and used to retrieve context, which is then fed into an LLM to generate a grounded answer.

- **Next.js Frontend Design:** On the frontend, implement a chat interface with an input box and chat history. Using **Next.js App Router** (if using Next 13+) or pages, you can create a React component that handles user input and displays a list of conversation messages (user and assistant). The Vercel AI SDK's `useChat` hook greatly simplifies managing this state and integrating streaming responses. It can automatically call your API route and handle server-sent events to update the UI token by token [14]. Ensure the UI remains responsive: use loading spinners or skeleton text while waiting for answers, and allow the user to cancel or send a new question even if a previous answer is streaming (if needed). For styling, frameworks like **Tailwind CSS** (used in Vercel's example templates [17]) or component libraries (Material UI, etc.) can help create a clean, developer-friendly design. Keep the design simple and focused: a chat window with scrollback, and perhaps an optional sidebar or dropdown to select document sets (if in future you have multiple knowledge bases).

- **Integration for Accurate Answers:** The integration between the UI and the knowledge base should emphasize **contextual accuracy**. Some strategies to achieve this:

- Always include the top relevant snippets in the prompt and instruct the model *not* to deviate from them. For instance, use a system message like: *"You are a documentation assistant. Answer only with facts from the provided content. If the answer is not in the content, say you do not know."* This reduces the chance of hallucination by confining the model to the retrieved text.
- Use a **re-ranking or filtering step** if possible: if you retrieve, say, 5 chunks, you might pass them all to the model, or you could first quickly analyze which ones are most likely to contain the answer (perhaps by a keyword overlap or a secondary similarity check). Libraries like LangChain can help with a "Stuff then refine" approach, but given the budget constraints, a simpler approach might be fine.

- Implement basic **citing** in answers: As mentioned, you can map each retrieved chunk to a source (like "Doc1, section 2") and then when the model answers, ensure it includes an indicator, or do a post-processing regex to append "[source: Doc1]" where appropriate. Many QA implementations simply instruct the model: "Include the source name in parentheses after each fact." This trains the model to output e.g., "According to *Doc1* (source 1), ...".

- **User Authentication (Azure AD via NextAuth):** Secure the chatbot behind login if needed using **NextAuth.js**. NextAuth supports Azure Active Directory as a provider, enabling enterprise single sign-on with just some configuration [18] [19]. You will need to register your app in Azure AD (as a multitenant or single-tenant app depending on your user base) and then provide NextAuth with the Client ID, Client Secret, and Tenant ID. Using the NextAuth Azure AD provider, you can have users authenticate via Microsoft and ensure that only authorized users (e.g., your organization's employees or students) can access the chat UI. Initially, **no advanced RBAC** is needed – meaning once a user is authenticated via Azure AD, they have the same level of access within the app as any other authenticated user. NextAuth will handle establishing a session (via secure cookies or JWT) and you can use its session data (which includes the user's email and AD details) to identify the user if needed. For now, you might simply use authentication to gate access (the entire app or at least the chat page checks `session ? <ChatUI/> : <SignIn/>`). Later, if FERPA or internal policy requires, you could implement role-based access control by reading claims (for example, user's group membership or roles in Azure AD) and restricting certain knowledge base content or features accordingly. NextAuth makes it easy to add such checks (via its callbacks or by using middleware to protect routes).

- **Frontend Best Practices:** Use Next.js built-in features for performance and maintainability:

- Leverage **API Routes** for server-side operations (embedding queries, database lookups, calling OpenAI). These run on the server (and in Vercel's case, as serverless functions) keeping your secret keys safe and allowing heavier computation off the client.
- Implement **caching** on the edge if possible for any static or semi-static content. While chat answers are dynamic per query, if you have certain constant data (like a list of available documents or categories), cache them in memory or using Next.js revalidation to avoid re-fetching.
- Ensure the app is mobile-responsive if needed (many users might interact on various devices). Use simple layouts that stack on small screens.
- Monitor errors on the front-end (use Vercel's analytics or Sentry integration) so you can catch if users encounter issues (like auth failures or API errors).
- Keep the dependency list lean to stay within budget on Vercel (free tier has function execution limits, etc.). For example, prefer the lightweight OpenAI SDK over installing the entire Azure SDK, unless needed. Similarly, if using LangChain, include only necessary modules.

Potential Challenges & Solutions: A common challenge is **maintaining answer accuracy**. Even with context, the LLM might sometimes fabricate an answer if the prompt isn't tight. A practical solution is to analyze the model's behavior and add guardrails: e.g., if no relevant context was found (vector search returns low similarity scores), have the backend detect that and respond with "Sorry, I don't know that." rather than asking the model, which could hallucinate. Another challenge is handling follow-up questions in a conversational way (context carry-over). This can be achieved by storing conversation history and including the last few QA pairs in the prompt. The Vercel AI SDK and Next.js can maintain session state or use the React state to accumulate history. However, this increases token usage, so you might limit history or summarize it. In terms of integration, making sure Azure AD auth doesn't disrupt the user experience is key – test the login flow thoroughly (NextAuth has a default UI you can use initially, or build a custom login page). Finally, if using serverless on Vercel, note that cold starts or request time limits might be an issue for long-running requests. Vercel serverless functions have a timeout (10 seconds by default). A single question might take ~5-15 seconds to answer (especially if using GPT-4). If you anticipate close to or above 10s consistently, consider using Vercel's Edge Functions (which can stream faster) or break the process: e.g., stream partial results. In many cases, GPT-3.5 on a short prompt will finish in under 10s, so it may be fine. Monitoring response times will be important; if needed, moving the heavy lifting to an Azure Function or a persistent server (which can handle longer processing) is an alternative.

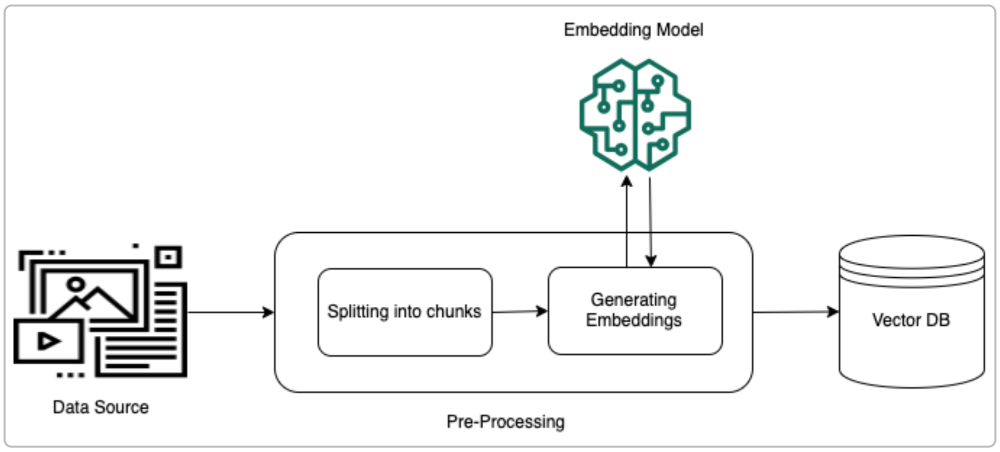
3. Text Chunking and Embedding Service

The heart of the RAG system is the text **ingestion and embedding service**. This service takes documents (from PDFs, HTML, etc.), breaks them into chunks, vectorizes those chunks, and stores them in a vector database for retrieval. Key recommendations:

- **Document Parsing and Chunking:** Use robust libraries to parse documents into clean text. **Unstructured.io's** `unstructured` Python library is a strong choice for handling various file formats. It is specifically designed to extract and transform content from complex formats (PDF, Word, HTML, etc.) into structured text suitable for AI applications [20]. Unstructured can, for example, pull out text from PDFs while preserving structure (paragraphs, headings) and output a normalized JSON or text that's ready for downstream use [20]. This saves a lot of preprocessing time by handling quirks of PDFs (weird encodings, multi-column layouts, etc.) behind the scenes. As the blog describes, it "streamlines the data preprocessing task" and converts hidden content in PDFs into AI-friendly form (like JSON or plain text for embeddings) [20]. In practice, you might use Unstructured's high-level API (like `unstructured.partition_pdf("file.pdf")`) to get a list of elements (title, narrative text, etc.), then join or further split those as needed.

If Unstructured is too heavy or not needed for some formats, more lightweight alternatives include: - **pdfminer.six**: A pure-Python PDF text extraction library. It can extract text and layout information from PDFs. You could use pdfminer to get all text and then split by pages or paragraphs. However, pdfminer can be slow on large PDFs and may require tuning to handle layouts. LangChain provides a PDF loader that wraps pdfminer for convenience [21]. - **PyMuPDF (fitz)**: An efficient library to read PDF files; often lauded as faster and more accurate in text extraction than pdfminer. PyMuPDF can extract page text easily and you can also get coordinates if needed. A Reddit discussion noted that PyMuPDF is one of the best open-source libraries for PDF text extraction, outperforming others like PDFPlumber or PyPDF in many cases. - **Microsoft Office docs**: If you have Word or PowerPoint files in the future, `unstructured` also handles those. Alternatively, libraries like `python-docx` for Word can extract text. For HTML pages (if some docs come from HTML dumps), **BeautifulSoup** is great to parse and strip HTML tags. - **EPUB/MOBI e-books**: Initially not required, but planning for future: EPUB is essentially a zip of HTML files. You can use libraries like `ebooklib` in Python to load EPUB and iterate through its chapters. Unstructured might add support for EPUB, but if not, a workaround is to convert EPUB to HTML or text (there are command-line tools like `ebook-convert` from Calibre). MOBI (Kindle format) is more proprietary; often converting MOBI to EPUB or PDF (via Calibre or Amazon's tools) then processing is the way to go. So, for future-proofing, design the chunking module to allow **file type handlers**. For example, have a function `process_file(file_path, file_type)` that routes to the appropriate parser for that type (PDF, DOCX, HTML, EPUB, etc.). This way adding MOBI support later is a matter of writing a new handler (perhaps using an external conversion if needed).

- **Text Splitting Strategy:** Once raw text is extracted from documents, it must be split into chunks that fit the vector model's context window and make semantic sense. The goal is chunks perhaps on the order of a few hundred words (or a few thousand characters) – large enough to contain meaningful info, but not so large as to dilute relevance or bust token limits when several are combined in a prompt. **LangChain** provides utilities like `RecursiveCharacterTextSplitter` which can split text while trying to respect sentence or paragraph boundaries. Another approach is **structure-based splitting**: if the docs have sections or headings, use those as natural chunk boundaries. For example, each top-level section of a guide could be one chunk. You might combine that with a max size rule (e.g., if a section is over 1000 words, split into subsections or paragraphs). It's also useful to allow a bit of **overlap** between chunks (like a sliding window) if you want to ensure that information that straddles a boundary isn't lost – though overlap increases total chunks and storage. The approach can be refined per content type: an API reference might be split per endpoint, whereas a narrative tutorial might be split per paragraph or subsection.



Document ingestion pipeline: raw data sources are parsed and split into chunks, then an embedding model converts each chunk to a vector, which is stored in a vector database for fast similarity search. [22] [23]

- **Embedding Generation:** For each text chunk, generate a vector embedding using a pre-trained model. A popular choice is OpenAI's **text-embedding-ada-002**, which produces a 1536-dimensional embedding and is renowned for good semantic representation at low cost. If using Azure, the Azure OpenAI service provides the same model. The cost is around \$0.0004 per 1K tokens, so embedding even thousands of pages is affordable (e.g., 1 million tokens worth of text is \$400; your documentation likely is far less). Alternatively, if wanting to avoid external API costs or dependencies, you could use open-source models (like sentence-transformers, e.g., `all-MiniLM-L6-v2` or other SBERT models). These can be run on CPU/GPU you manage. Open-source embeddings are free to run, but require provisioning hardware and possibly not as high-quality semantic matches as OpenAI's model. Given the budget and scalability, using the hosted API might be simpler initially – you pay per use and avoid maintaining a GPU server. Just be mindful of rate limits; Azure OpenAI and OpenAI API have rate limits per minute, but those likely won't be an issue with weekly batch processing and moderate query load.

- **Vector Database Storage:** Choose a vector store that is cost-effective yet scalable:

- An excellent low-cost approach is to use **PostgreSQL with the pgvector extension**. This allows you to store embeddings in a regular Postgres table and use indexed vector similarity search. Vercel's RAG template demonstrates using Postgres (hosted on Vercel or elsewhere) with an ORM (Drizzle) to manage the vectors [24] . The advantage is you can piggyback on a general-purpose database (which you might already need for metadata or user data) and keep costs down (a small managed Postgres instance or even Azure Postgres Basic tier). pgvector supports approximate nearest neighbor search and can handle thousands to millions of vectors if properly indexed.
- **Pinecone** is a specialized managed vector DB with a generous free tier (as of current offerings, free tier allows a limited number of indexes and memory up to a certain limit). Pinecone is very easy to integrate (with their JS/Python client) and highly optimized for vector operations. However, beyond

the free tier it can become costly. If your document corpus is modest (say a few thousand chunks), you might fit in the free tier initially [25] . Pinecone's advantage is you don't worry about infrastructure at all.

- **Qdrant** or **Weaviate** are open-source vector databases that you can self-host. For example, you could run Qdrant as a Docker container on an Azure VM or use their cloud (Qdrant has a free tier too). Self-hosting one of these on a small VM might cost \$20-30/month but would give you full control and typically very fast performance (they're written in Rust, etc.). This might be preferable if you anticipate growth (since you won't be locked into a free tier limit).
- **Azure Cognitive Search** now supports vector search as well, and integrates with Azure OpenAI for "retrieval augmented generation" pipelines. It's a convenient option if you want a single Azure service to handle indexing and searching your docs (including hybrid vector + keyword search) [26] [27] . The downside is cost; Azure Search units can be pricey (likely >\$100/month for any substantial storage), so it may not fit the strict budget. It's something to consider long-term, especially if you need enterprise features or scaling that outgrows a single Postgres or small vector DB.

Metadata and Source Tracking: Whichever vector store you use, **store metadata alongside embeddings**. At minimum, each vector should have fields for the source document identifier (e.g., file name or URL), and maybe a section or page number. This is crucial for later providing citations and for filtering. For instance, your vector table could have columns: `id, doc_title, section_heading, page_number, chunk_text, embedding_vector`. Some vector DBs (like Pinecone, Weaviate) allow arbitrary key-value metadata with each vector. By storing the `doc_title` or `source_url`, you can later retrieve that and present "(Source: Doc ABC, page 10)" with the answer. It also allows queries like "only search within Doc A" if you implement a filter (many vector DBs support metadata filtering).

- **Citation Retention:** To maintain citations through the RAG process, implement a strategy from the start:
- **Prompt-based citations:** One simple way is to inject the source name into the text chunk itself when feeding the model. For example, when assembling the prompt context, instead of just raw snippet text, include a reference tag: "

$$Doc1p.2$$

: <snippet text>". Then ask the model to use those tags when answering. This often leads the model to say something like "According to Doc1 p.2, ...". You can post-process to format it nicely.
- **ID referencing approach:** Another approach is to assign each snippet an ID (like a short alphanumeric or a number). Provide the model a list of snippets with IDs, and instruct it to cite using those IDs. LangChain's documentation suggests tool usage or direct prompting to have models cite documents by ID [28] [29] . For instance, you could give the model: "Context:\n(1) [text of snippet1]\n(2) [text of snippet2]\nQ: ...\nA: ... please cite sources by number." This way the model might answer: "The answer is XYZ [1]", which you can map back to the document.
- **Out-of-band citations:** Alternatively, the back-end can handle citations by keeping track of which chunks were used. If the model's answer contains a particular fact, we know which chunk had that fact (since we retrieved top K, presumably one of them). Some implementations choose to simply always cite all retrieved documents at the end of the answer. For example: "*Answer content...* \nSources: Doc1, Doc2." This guarantees citations but is less precise. A more precise but complex method is a second pass where you feed the answer and ask another model (or heuristic) to identify which source best supports each sentence, but that's likely overkill initially.

Regardless of method, *preserving the connection between answer and source is crucial for user trust*. By storing metadata as mentioned, you make this feasible. In the design, emphasize that each chunk knows where it came from. Then it's a matter of either prompting or logic to propagate that to the final answer. You might start with a simple solution (e.g., always append the doc title), and refine as users demand more clarity.

- **Uploading New Documents (Architecture):** Provide a mechanism (likely a web UI in the app) for admins or authorized users to upload documents (PDFs, etc.) to the knowledge base. This could be a page in the Next.js app that allows file upload (using a file input or drag-and-drop). The uploaded file can be sent to an API route (or directly to Azure Blob Storage). Upon receiving a new file, you have a few options for processing:
- **Inline processing:** The API route that receives the file can immediately run the parsing, chunking, and embedding steps, and update the vector store. This is simpler, but be cautious: large files could make the request run long or even time out. If going this route, consider increasing any timeouts and providing user feedback (e.g., "upload in progress…").
- **Background processing:** A more robust method is to offload processing to a background worker or function. For example, the Next.js API route could store the file (to disk or cloud storage) and enqueue a job (perhaps by calling a separate Azure Function or posting a message to a queue like Azure Storage Queue or RabbitMQ). A background worker service (which could be the same one as the crawling pipeline, if it's running continuously, or a separate lightweight service) would pick up the job, process the file, and then mark it done. This decouples user uploads from heavy processing so the user isn't stuck waiting on a HTTP request. If using Vercel, note that long processing isn't ideal on serverless, so background is better. You can use tools like **BullMQ** (Redis-based queue) or simply trigger your scraping script with parameters.

- **Architecture Suggestion:** You might unify the crawl pipeline and upload processing by having a single "ingestion service." For instance, a Python script (or FastAPI app) that can be triggered to ingest content either from a URL (crawling) or from an uploaded file path. When a user uploads a file, your Next.js could call this service's API (passing a reference to the file in blob storage or an uploaded binary). This way all chunking/embedding logic resides in one place (Python service), which is easier to maintain. This service could run on a small VM or container in Azure. The cost for a small container (e.g., Azure Container Instance or an App Service) could be low, and it can be kept off (scaled to zero) until needed if using something like Azure Functions with a Docker (although not trivial). For budget, maybe a tiny VM that runs 24/7 (~\$15/month) could handle occasional jobs.

- **Metadata Enrichment:** In addition to basic source info, consider tagging chunks with other metadata that could be useful. For example, *document category* or *product version* (if your docs span multiple products or versions). This allows filtering: e.g., if the chatbot has a dropdown "Product A vs Product B", you could filter the vector search to only that product's docs. Initially, you might not need this, but designing the schema to allow it (even if blank for now) is forward-thinking. Another example: a chunk could store `created_at` timestamp or a version number of the document. This can aid in reproducibility (we can identify which version of a doc the answer came from).

Potential Challenges & Solutions: One challenge is **handling document changes**. If a doc is updated, the chunks and embeddings for that doc should be updated in the vector store. You will need to delete or overwrite the old chunks. It's wise to have a unique ID per document (e.g., a path or GUID) and perhaps a version number. When re-ingesting, you can remove all vectors with that doc ID and then add new ones. This ensures no stale data lingers. Another challenge is **quality of chunking**: if chunks are too large, the model might get irrelevant text; if too small, important context might be split up. A practical solution is to experiment – try different chunk sizes on a sample Q&A and see which yields better answers. A hybrid approach is also possible: store both small passages and some larger summaries. For instance, you could generate an embedding for an entire doc (or section) as well, and during retrieval do a two-stage: first find relevant doc(s), then within those find specific chunks. This could improve precision and speed if the number of docs is large. However, that adds complexity, so weigh it against current needs (with only thousands of docs, a single-stage similarity search on all chunks might be fine).

Handling **media or non-text content** in documents is another issue. If PDFs have images or diagrams critical to understanding, pure text extraction might miss context. The system as designed is text-centric. In the future, you could integrate OCR for images or add descriptions for diagrams manually. For now, just be aware of this limitation and maybe include a note in documentation that the assistant might not handle image-based info. If needed, one could store image captions or figure descriptions if the docs provide them.

Finally, ensure that the **embedding service is reproducible and testable**. That is, given the same input file, it should consistently produce the same chunks and vectors (unless the file changed). This can be tested by running the pipeline twice and diffing outputs. Deterministic behavior (no random shuffling) is important for data integrity.

4. Data Integrity and Reproducibility

Maintaining data integrity means that your knowledge base contents (documents and embeddings) are trustworthy and traceable. Reproducibility means you can regenerate or track the state of the system at any point in time. Recommendations in this area:

- **Version Control for Data:** Treat your documentation data as an evolving dataset that needs version control similar to code. Use Git or a data versioning tool to keep copies of original documents (and possibly processed texts). For example, you could store all raw documents (HTML pages, PDFs, etc.) in a **GitHub repository** or an Azure Blob Storage container with versioning enabled. Each weekly crawl could correspond to a git commit or a storage snapshot. This way, if a question arises about what data was present at a certain time (e.g., *“What did the bot know last month?”*), you have the historical data. For large files, Git LFS or DVC (Data Version Control) can be used to manage them. Tools like **DVC** can version large datasets (like a collection of embedding vectors or text files) efficiently [30]. DVC would let you associate a version of the dataset with a git commit of the code, ensuring you know which embeddings go with which code version [30]. Initially, this might be overkill, but adopting even a simple convention (like date-stamping data dumps) helps.

- **Embedding Versioning:** The embedding model itself may change over time (for instance, OpenAI might release `text-embedding-ada-003` in the future, or you might switch to a different model). Changing the embedding model or parameters is a **breaking change** for the vector index – vectors are not comparable across different model types or dimensions. Plan for this by version-tagging your embeddings. Include the model name and version as part of your metadata or even as part of index name. For example, keep indices like `embeddings_v1` vs `embeddings_v2` if you do migrate. If you anticipate an upgrade that changes vector dimensionality, you might run two indices in parallel (as a sanity check) and have a way to fall back if needed. This is akin to semantic versioning in software: a new embedding model that changes vector size or distribution would be a major version bump because it's incompatible [31]. Minor version changes (like using the same model but maybe a different chunking strategy) also should be tracked, though they might be compatible with existing vectors (if dimension same, you could still store them together, but it's good to know which chunk was created with which method). Maintaining clear documentation of these versions and changes is important [32]. For instance, keep a CHANGELOG for your data pipeline: *"2025-05-18: Ingestion v1.2 – switched embedding to ada-002, re-embedded all data"* etc.

- **Consistent Chunking and Embedding Output:** To ensure reproducibility, the pipeline should be deterministic. That means if you run it on the same input data with the same code version, the output chunks and embeddings should be identical. Avoid randomness in splitting (most splitting algorithms are deterministic based on content). If you use any processes that could be non-deterministic (for example, an ML model for summarizing or a multithreaded process that could order outputs differently), add controls: sort outputs, fix random seeds where applicable. This consistency allows you to rerun the pipeline and verify you get the same result, which is a hallmark of reproducibility. It also simplifies debugging differences – if outputs differ, something truly changed (either input data or code). Use hashing or checksums: for example, after chunking a document, you could hash each chunk text. If you re-chunk later and the hashes differ, you know the chunk content changed (which might indicate a code change effect). Storing these hashes can also serve as a quick integrity check to ensure no corruption in your vector store (e.g., compare a chunk's current hash to a previously stored hash for that chunk's ID).

- **Provenance Tracking:** It's important to trace each answer back to the source data version. Implement logging in the QA pipeline: whenever an answer is generated, log which document IDs and chunk IDs were used to form that answer. These logs (even if just kept internally) mean you have an audit trail: if someone says *"Your bot gave me wrong info,"* you can check and see exactly which source text it was relying on. This also helps if later the document is updated – you could potentially inform that *answers prior to update X may be outdated*. If needed for compliance or future debugging, you could store the entire prompt that was sent to the LLM (with context and question). However, be careful with that if the context might include sensitive data by then (for now it's public docs, so fine). At least storing the list of sources used is a lightweight alternative.

- **Reproducing the Environment:** Use consistent environments for embedding generation. If you run a local Python for embeddings and then later run on a different machine, slight differences in floating point or library versions usually don't matter for embedding vectors (since the model outputs are fixed given the same input). But ensure you use the same model and API version. For instance, OpenAI might sometimes update models behind the scenes – you can mitigate surprises by specifying model version/date if the API allows, or by promptly re-embedding everything if an update occurs that slightly changes embeddings. If using open-source models, pin the model version (don't frequently pull the "latest" model weights without testing). Containerize the pipeline (with Docker) so that the same image can be used each run, avoiding issues like one run using a different library version. This also helps with **long-term maintainability**: a new developer or a future you can run the pipeline container and get the same behavior.

- **Backups and Fallbacks:** Consider taking periodic backups of the vector database (if possible). For example, if using Postgres, do a dump of the embedding table every month. If using Pinecone, maybe export the vectors (they have an API to fetch vectors) to have a copy. This guards against any data loss or corruption – you could rebuild by reprocessing documents too, but backups might be faster. Since FERPA might be in scope later, having backups also means you can ensure data compliance or roll back to a previous state if something goes wrong with an update (for instance, if an update accidentally ingested some sensitive data it shouldn't, you can revert).

Potential Challenges & Solutions: A challenge in versioning is when to **re-embed** everything vs. incrementally update. If a small change in code is made (say you improved the text cleaning to remove stopwords), do you reprocess all docs for consistency or only new ones with the new method? It's often better to reprocess for consistency, but that could be costly. A pragmatic solution is to bundle changes and do occasional full re-indexing (maybe when you accumulate enough changes or if there's a new embedding model that significantly improves results). Communicate these changes as part of internal documentation so everyone knows which version of the KB is live. Another challenge is maintaining **schema changes** – e.g., you decide to add a new metadata field or change how metadata is structured. If using a relational DB, migrations are needed. This is where an ORM or migration tool helps. If using Pinecone, you might not have schema beyond metadata keys, which is simpler.

From a reproducibility/testing standpoint, you might want to simulate queries in a test to see if answers remain consistent after an update. For example, have a set of sample questions and expected sources; when you update the pipeline, run these queries to see if the same sources are retrieved and the answers still make sense. If something wildly changes, you've caught a potential issue. This kind of regression testing for an AI system is not exact (because the model could give variations), but checking the retrieval stage is fully deterministic given same embeddings – you can ensure that part is stable.

Lastly, when FERPA or other compliance is considered in the future, having this rigorous version control and provenance will make it easier to answer questions like *"What data did the system train on or use on this date?"* – you'll have the records. It also helps in potentially removing data: if a document must be removed for compliance, you know exactly which vectors to delete.

5. Scalability and Performance

The system should scale to handle a growing knowledge base (thousands of documents) and concurrent users (~200+ users), all while keeping costs manageable. Here we outline infrastructure and optimization strategies to achieve **fast responses (goal: <20s per query)** within the **\$100/month budget**:

- **Infrastructure Architecture (Cost-Conscious):** To minimize costs, leverage as many free or low-tier services as possible:
- Host the **Next.js frontend on Vercel**. Vercel has a generous free plan for hobby projects, which might suffice initially. The main cost consideration with Vercel is if your app makes heavy use of serverless functions (API routes) beyond the free quota. Monitor usage; if it's high, consider upgrading to a small Pro plan or migrating some serverless work to Azure Functions (which also has a free grant and can be cheaper for high volumes).
- Use **GitHub for code and CI** (free) and possibly for the weekly automation as discussed (GitHub Actions free tier should cover a weekly job easily).
- For the **vector database**, an open-source self-hosted solution on a small VM is likely the cheapest path if the free tiers of managed services are insufficient. For example, an Azure B1s Linux VM (1 vCPU, 1 GB RAM) costs only around \$10-15/mo. You could run a Postgres with pgvector or a Qdrant instance on it. Thousands of documents (let's say 5k docs, split into ~50k chunks) and 1536-dim vectors can fit in a couple of GB of RAM. A 1GB RAM VM might be tight at that scale (depending on

index type), but you could step up to a 2GB instance for ~$20-25/mo. This is still within budget. If using Postgres, you also get a place to store metadata and user data if needed. Ensure to tune the DB (enable vector index, etc.). If using Pinecone's free tier, check the limits (e.g., perhaps 5k vectors limit); that could be a zero-cost solution until you hit the limit [25].

- The **embedding and question-answering LLM** will likely be via API calls to Azure OpenAI or OpenAI. This is a usage-based cost. To stay under budget, use the cheaper models: GPT-3.5 Turbo for chat (~\$0.002 per 1K tokens) and Ada for embeddings (~\$0.0004 per 1K). These rates mean a single question that uses, say, 2K tokens in and out (~1500 tokens answer, 500 tokens prompt) is about \ $0.004, less than half a cent. 200 such queries is \$0.80. If each of 200 users asks one question, that's under \$1 – very cheap. The cost can add up with more usage, but even 1000 queries is \$4. So the LLM usage might not be the largest cost unless users ask thousands of questions every day. That said, to be safe, implement some usage controls: e.g., limit the maximum tokens or put a cap if needed. GPT-4 is much more expensive (~15x), so perhaps restrict to GPT-3.5 unless a specific query really needs GPT-4 (maybe offer an option for admin).

- If you foresee the need for offline or unlimited use, exploring an **open source LLM** hosted on Azure (like a fine-tuned Llama-2) could eliminate API costs, but running such a model 24/7 likely busts the budget in compute. So it's a trade-off: API costs are predictable and scalable at low volumes, hosting a model is a fixed cost whether it's used or not.

- **Handling Thousands of Documents:** Vector search scales sub-linearly with approximate algorithms. In practice, vector DBs like Qdrant, Weaviate, or Pinecone use HNSW or similar, where searching among 50k vectors is very fast (a few milliseconds) if properly indexed. Ensure that the vector index is configured to use an approximate algorithm for speed. If using Postgres pgvector, you might rely on an IVF index or just do exact search with index if small. In worst case, exact search on 50k vectors of 1536 dims could be a few hundred milliseconds – possibly acceptable. Monitor the retrieval time; if it's creeping up, consider adding an index or switching to a dedicated vector engine which is optimized for this. Also, as the number of documents grows, consider sharding your index or categorizing (like if you have very distinct topics, you might maintain separate indexes per topic to narrow search – this can improve both speed and relevance).

For storage, thousands of docs with vectors will occupy some space (each vector is 1536 floats, ~6KB if float32, less if using float16 or compressing). 50k such vectors ~ 300 MB of raw data. That fits easily on a small disk. If on Pinecone or cloud, you pay for the vector count typically, but again, should be manageable initially. The main point: the architecture can handle scaling in *data size* by either beefing up the vector store (more memory or switching to a cluster as needed) and *data organization* (maybe adding metadata filters as above to partition queries).

- **Concurrent Users (200+):** Concurrency affects mainly the query-serving component. With Next.js on Vercel, each request (question asked) can spawn a serverless function instance. Vercel will automatically scale out as requests come in, so theoretically it can handle 200 concurrent requests by running 200 function invocations in parallel. The limits you need to watch are:
- **OpenAI API rate limits:** If 200 queries hit at exactly the same second, you are making 200 embedding requests and 200 chat requests simultaneously. OpenAI's API has per-minute and per-second limits (for example, maybe 150k tokens/minute for certain accounts). 200 queries might be fine, but you might need to request rate limit increases if you consistently run at high QPS. Alternatively, queue or throttle at your end – you could implement a simple queue in the API: if >N requests in flight, wait. However, for user experience, it's better if you don't have to throttle.

- **Database connections:** If using Postgres, 200 concurrent requests means 200 concurrent DB queries for retrieval. Postgres can handle that if configured, but a tiny instance might struggle. Use a connection pool and set max connections thoughtfully. You might run PG with max 100 connections and the rest wait a bit. But since vector search is quick, the contention might not be severe. Another idea: use a read-replica or even keep an in-memory copy of vectors in the function (not really feasible in serverless, each function invocation is isolated).
- **Memory/CPU on serverless:** Each Vercel function might use some memory for the model responses. 200 concurrent might push the aggregate memory, but since each is separate, it should be fine as long as each within limits (~128MB or 256MB per function by default). It might increase usage costs if you go beyond free, but likely still okay.

Overall, this architecture is quite horizontally scalable because no single stateful server is handling all users – the work is distributed across serverless instances and the database. The vector DB or Postgres is one centralized component that must handle concurrent queries; ensure it's hosted on sufficient hardware. If on a small VM, 200 simultaneous queries might cause high CPU usage for a brief time (especially if computing dot products on 50k vectors each time). If that becomes an issue, you may need to scale up the VM or move to a managed service that scales (like a larger Postgres or a scalable vector DB). Keep an eye on metrics – e.g., CPU, memory of the vector DB machine, and query response times.

- **Optimization for <20s responses:** Achieving sub-20 second answers involves optimizing both **retrieval** and **generation**:
- *Retrieval speed:* With a proper index, retrieving top-k documents should be very fast (tens of milliseconds). Ensure to only retrieve a reasonable number of chunks (commonly 3 to 5). More chunks = longer prompt and potentially more irrelevant info to confuse the model. Empirically, 3 top chunks often suffice if your chunks are well-sized and your embeddings are good. If using an approximate search, tune it to balance speed vs accuracy (e.g., the HNSW `ef` parameter – higher means more accuracy but slightly slower; you might start moderate and adjust if you see it missing relevant stuff).
- *Prompt construction:* Keep the prompt concise. Don't include boilerplate text or lengthy instructions beyond what's needed. Every token the model sees or generates counts toward latency. If you have 5 chunks of 1000 tokens each, that's 5000 tokens + user question + instructions, which might be near the model's context limit and certainly will slow generation. If you find responses are slow, consider retrieving fewer or shorter chunks. You could also **summarize** or truncate chunks: for example, if a chunk is very large (maybe your chunking strategy allowed a 1000-word chunk), you might truncate it to the first 200 words for the prompt (assuming the most relevant content is at the top). This is a heuristic that risks missing info, but can cut down on prompt length.
- *Model choice:* **GPT-3.5 Turbo** is much faster (and cheaper) than GPT-4. GPT-4, while more accurate, often has a latency of 10-30 seconds for a response even with moderate length input. GPT-3.5 can often answer in a few seconds. So for performance, use GPT-3.5. Some apps use GPT-4 only for certain queries or when user specifically opts in. Given budget, probably stick to 3.5 for all. If even 3.5 is slow, you could consider smaller models (like if you had a local LLM with lower latency, but currently OpenAI's 3.5 is hard to beat on speed/quality trade-off).
- *Streaming:* Enable streaming of the answer to the user. This doesn't reduce the time to final token, but it significantly improves perceived performance. If the first part of the answer appears in 2 seconds and the rest streams in over the next 8 seconds, the user is already reading the answer within the 20s window. Vercel's AI SDK and OpenAI API both support streaming easily. Ensure your Next.js API responds with a stream (set proper headers, etc., which the SDK does for you). On the front end, the `useChat` hook will populate the message as it arrives [14].

- *Parallelization:* Within a single query, there's not much you can parallelize because you have to embed, then search, then call LLM. However, you can parallelize *across* queries (which is what concurrency is). One micro-optimization: embed the user query and start the LLM call **in parallel** with the vector DB lookup – this is only possible if you use something like OpenAI functions or tools (where the model could do the lookup as a tool call) or if you optimistically generate an answer without context (not useful). Generally, you need the retrieved docs before calling LLM, so that step is sequential. But since vector search is fast, it's not the bottleneck. The LLM generation is the bottleneck, and that you can't really parallelize for one query (it just has to generate tokens one by one). So focus on making that step faster via model choice and prompt length reduction.
- *Content caching:* Implement **caching** layers where feasible. For example, **semantic caching**: cache the answers to previous questions and even embeddings [33] . If the same (or very similar) question is asked again, you can serve the cached answer instantly instead of recomputing [33] [34] . Similarity can be determined by comparing the new question's embedding to past question embeddings stored in a cache (which is essentially a vector search among asked questions!). This is an advanced feature that can drastically cut cost and latency for repeat questions. At 200 users, it's likely many will ask similar things (especially if documentation has FAQs). A simpler cache: if user asks identical text as a previous user, just reuse answer for some time window. You could store a dictionary of recent Q->A. This has to be done carefully to avoid serving outdated info if docs update, but for docs that change weekly, a short-term cache (hours or days) is usually fine.

- *Rate limiting & queuing:* If maintaining sub-20s response during heavy load is challenging (perhaps if 200 all ask at once, some might experience slowdown due to API queueing), consider a gentle queue on the client side. For instance, if a user spams multiple questions, handle one at a time. Or globally, if your system can only process, say, 100 concurrently with good speed, you might serve an error or a "please wait" for the 101st until one finishes. This is a last resort if capacity is limited. Given the architecture, probably not needed initially, but keep it in mind if usage spikes unpredictably.

- **Trade-offs and Acceptable Sacrifices:** Some trade-offs to consider in balancing performance, cost, and accuracy:

- *Accuracy vs Speed:* Using fewer context chunks or a smaller model improves speed but might reduce answer accuracy or completeness. You need to decide what's acceptable. For a documentation assistant, it's usually better to prioritize accuracy (no hallucination, correct info) over a few extra seconds. However, if users are highly sensitive to delay, you might lean towards brevity. One middle ground is to limit the answer length – extremely long answers take longer and use more tokens (cost). Often an answer can be concise. You can instruct the model to be brief if needed.
- *Cost vs Thoroughness:* To stay under budget, you might restrict how often the pipeline runs or how many documents you index. If docs are huge (like thousands of pages), you might initially index summaries of them rather than full text to save on embedding cost and search speed. This sacrifices detail for cost. As an example, if FERPA compliance or proprietary data is needed later, you might need a private secure environment which could raise cost – so maybe you'd then limit usage or require more targeted queries rather than open-ended ones to reduce LLM usage.
- *Complexity vs Maintainability:* Adding a lot of optimization layers (caches, parallel processes, etc.) can complicate the system. Given a small team and budget, it may be acceptable to have a simpler design that might not be fully optimal but is easier to maintain. For instance, running everything on one medium VM (doing both vector store and hosting an API) is simpler to maintain than splitting into multiple microservices – but one big VM is a single point of failure and could be slower. Decide based on team skill and whether the user load really necessitates the more complex setup.

- *Immediate Response vs Accuracy:* If truly <20s is hard to meet under load, you could implement an **asynchronous response** pattern: the user asks a question, and if it's going to take longer, the system responds with something like "Your answer is being prepared, please check back in a moment." Then finalize the answer and show it. However, this is generally a poor UX for chat unless the delays are very high. With streaming, we aim to avoid this entirely.

Long-Term Scalability: As usage grows, monitor which part becomes the bottleneck: - If the **vector DB** struggles (too many vectors or queries), consider moving to a managed scalable solution (like Pinecone or a bigger Qdrant cluster). - If the **LLM cost** grows (lots of questions daily), you might explore fine-tuning a smaller model on your data or using retrieval with a model like GPT-3 (cheaper but maybe less accurate) for some queries. Or use a hybrid: for very frequent simple questions, perhaps pre-compute answers (like an FAQ list that the bot can directly respond from without calling the LLM). - If concurrency grows beyond a few hundred (say thousands of users), you might need to implement user-specific rate limits or quotas to control cost and ensure fairness. Azure AD integration can help here (you can identify the user and maybe restrict heavy users or require certain roles for high volume).

Maintainability & Monitoring: Use logging and monitoring to keep the system healthy. Azure Application Insights or Vercel's monitoring can track response times and failure rates. Set up alerts if the response time goes above, say, 20s on average or if error rate spikes. This will let you intervene (maybe the embedding service is down or out of memory). A well-maintained system can then be scaled up by just increasing plan sizes or adding instances when needed, which is relatively straightforward on cloud platforms.

Conclusion: The proposed RAG system combines automated data ingestion, a Next.js-based chatbot UI, and a vector search backend to deliver accurate answers from documentation. By carefully choosing open-source tools (Scrapy, Unstructured, pgvector) and cloud services (Azure AD, Vercel) with cost in mind, the design stays within budget without sacrificing capability. We addressed each aspect – from crawling docs to delivering an answer – with an eye toward scalability, reliability, and future requirements (like compliance). By following these recommendations and best practices, an experienced development team can implement a maintainable RAG solution that provides users with quick, context-rich answers and can evolve with growing data and user demands.

Sources: The recommendations above draw on current best practices and tools as of 2025, including insights from official documentation and community expertise for web scraping, AI SDK usage, and vector databases. Key references include comparisons of scraping tools [3] [1], discussions of scheduling approaches [10] [11], Vercel's example RAG implementations [16] [35], Unstructured's documentation processing capabilities [20], and techniques for cost optimization like semantic caching [33] [34], among others.

---

[1] [2] [3] [4] Scrapy vs. Beautiful Soup for web scraping
https://blog.apify.com/beautiful-soup-vs-scrapy-web-scraping/

[5] [6] [7] web crawler - Incrementally crawl a website with Scrapy - Stack Overflow
https://stackoverflow.com/questions/37286480/incrementally-crawl-a-website-with-scrapy

[8] [9] How to perform Incremental Web scraping
https://stabler.tech/blog/how-to-perform-incremental-web-scraping

[10] [11] [13] Why Airflow? Can’t I just go with Crontabs? | by Suryanarayanan | Medium
https://medium.com/@suryanarayanan035/why-airflow-cant-i-just-go-with-crontabs-40dbccbbe15e

[12] Periodic Tasks — Celery 5.5.2 documentation
https://docs.celeryq.dev/en/stable/userguide/periodic-tasks.html

[14] [15] [17] [24] RAG with Vercel AI SDK
https://vercel.com/templates/next.js/ai-sdk-rag

[16] [25] [35] Pinecone - Vercel AI SDK Starter
https://vercel.com/templates/next.js/pinecone-vercel-ai

[18] [19] Azure Active Directory | NextAuth.js
https://next-auth.js.org/providers/azure-ad

[20] How to Process PDFs in Python: A Step-by-Step Guide – Unstructured
https://unstructured.io/blog/how-to-process-pdf-in-python

[21] PDFMinerParser — LangChain documentation
https://python.langchain.com/api_reference/community/document_loaders/
langchain_community.document_loaders.parsers.pdf.PDFMinerParser.html

[22] [23] How Amazon Bedrock knowledge bases work - Amazon Bedrock
https://docs.aws.amazon.com/bedrock/latest/userguide/kb-how-it-works.html

[26] [27] RAG and generative AI - Azure AI Search | Microsoft Learn
https://learn.microsoft.com/en-us/azure/search/retrieval-augmented-generation-overview

[28] [29] How to get a RAG application to add citations | LangChain
https://python.langchain.com/docs/how_to/qa_citations/

[30] [31] [32] How do you version and manage changes in embedding models?
https://milvus.io/ai-quick-reference/how-do-you-version-and-manage-changes-in-embedding-models

[33] [34] Building a Cost-Optimized Chatbot with Semantic Caching | Databricks Blog
https://www.databricks.com/blog/building-cost-optimized-chatbot-semantic-caching