ChatGPT

RAG-Based Conversational AI for Invoices – Architecture and Implementation

Overview and Requirements

Building a Retrieval-Augmented Generation (RAG) conversational assistant for business invoices requires combining document processing (OCR and parsing) with an LLM-driven Q&A system. The goal is to let users ask natural language questions about invoice data and get accurate answers grounded in the documents. We target a **lightweight MVP** that runs locally on a MacBook Pro M3 (leveraging Python + PyTorch), using **open-source tools** for easy development and future extensibility. The solution will ingest common invoice formats (PDFs, scanned images, Excel spreadsheets), extract and vectorize their content, and provide a conversational API (no GUI, API only) for querying this content.

Key requirements include:

- **Robust Document Ingestion** – Reading heterogeneous invoice formats and extracting text *and structure* (e.g. identifying totals, dates, line items).
- **Vector Store & Retrieval** – Embedding document text/fields and storing vectors for similarity search, enabling relevant content retrieval per query.
- **LLM Integration** – Using an LLM (local or API-based) to generate answers from retrieved context (RAG) in a conversational manner.
- **Conversational Interface (API)** – Exposing endpoints (e.g. via FastAPI) to accept user questions (and possibly maintain context across turns), returning answers.
- **Minimal Infrastructure** – Favor a simple, local-first tech stack (e.g. using local vector DB, open-source OCR/LLM) that can grow into a scalable SaaS (with cloud deployments, multi-user support) later.
- **Extensibility** – Use modular, open-source components (e.g. RAGFlow engine, Pathway/LLM-App templates, DeepSeek-OCR, LangChain, etc.) to minimize custom code but allow future customization.

Below we analyze candidate tools/frameworks and then present a recommended architecture, step-by-step implementation plan for a PoC/MVP, and an upgrade path toward a SaaS solution.

Tools & Frameworks for RAG Invoice QA

RAGFlow – Turnkey RAG + Agent Engine

RAGFlow [1] is an open-source engine providing an end-to-end RAG workflow with built-in agents and context management. It offers a streamlined pipeline to ingest and semantically index data, then answer queries with an LLM. RAGFlow's notable features:

- **Ingestion Pipeline:** It can ingest and **cleanse multi-format documents** (Word, Excel, PDF, images, etc.) and convert them into rich semantic embeddings [2] [3]. This suits our need to handle PDFs,

scans, and spreadsheets. RAGFlow's "deep document understanding" modules (e.g. MinerU, Docling parsers) interpret complex layouts and can even handle **scanned copies and images** [3].
- **Hybrid Search:** RAGFlow supports high-precision retrieval by combining **vector similarity, BM25 full-text search, and custom re-ranking** for best results [4]. This can improve accuracy in finding the right invoice content ("needle in a haystack" retrieval [5]) and reduce LLM hallucinations by always grounding answers in retrieved text.
- **Agentic Capabilities:** Beyond basic Q&A, RAGFlow has an integrated agent orchestrator. It supports tools and multi-step reasoning – e.g. one could build an agent that checks calculations or queries multiple sources as part of answering. This is advanced for MVP, but RAGFlow makes it available if needed (with templates for common agent behaviors [6]). It even supports long-term **memory for agents** (added in latest updates [7]), which could maintain conversational context.
- **APIs and Demo**: RAGFlow provides intuitive APIs and even a demo chat interface [6]. You can run it as a local service (or via Docker) and call its endpoints for ingestion and query. This potentially saves you from writing a lot of glue code – you configure RAGFlow and leverage its API as your backend.

Usage in this project: RAGFlow could serve as the *core engine* powering the invoice Q&A system. For a quick PoC, you might configure RAGFlow to ingest your invoice files (point it to a folder or supply files via API) and then use its query API to handle questions. Because it natively handles multi-format docs (including Excel and scanned PDFs) [3], it can simplify ingestion. Its hybrid search and context layering features would enhance answer reliability. However, RAGFlow's breadth means a bit of a learning curve – you'd need to follow its docs for configuration (YAML/JSON settings for data sources, which embeddings/LLM to use, etc.) and possibly install large models for the "deepdoc" parsing. For an MVP on Mac, it's feasible (it only needs >=4 CPU cores and 16GB RAM to start [8]), but consider that it's a comprehensive system. If you prefer more manual control or lighter setups, you might use RAGFlow selectively (e.g. only its ingestion and retrieval components) or opt for simpler frameworks like LangChain.

LLM-App (Pathway) – Real-Time RAG Pipeline Templates

LLM-App is a collection of ready-to-run application templates for LLM + RAG scenarios (open-sourced by Pathway.com, which provides a Python stream processing framework). Essentially, it provides pre-built pipelines (in YAML/Python) that **sync data sources, embed documents, and serve answers via API** [9]. Notable aspects for our use case:

- **Streaming Document Indexing:** LLM-App's templates use Pathway's incremental computation engine. You can declare data sources (e.g. a local folder, Google Drive, SharePoint, etc.) in a config, and the pipeline will continuously monitor and index documents in real-time [10]. For example, the "Question-Answering RAG App" template watches a local `data/` directory for documents, parses and chunks them, and updates a vector index whenever files are added or changed [10] [11]. This is great for a SaaS scenario where new invoices are ingested regularly (no need to manually re-run indexing). In an MVP, you could start with a simple one-time ingestion (or polling every few minutes), which LLM-App supports out-of-the-box.
- **Document Parsing and Embedding:** The default pipeline uses IBM's **Docling** model for document parsing [12] – an open-source library that focuses on extracting structured text and layout from documents (especially PDFs). Docling can handle complex layouts, tables, etc., complementing OCR by providing semantic structure [13]. By default, parsed documents are then chunked (via a `TokenCountSplitter`) and embedded using an OpenAI embedding model [14]. (The template uses `OpenAIEmbedder` by default, but you can **substitute your own embedder** [15] – e.g. a local SentenceTransformer – to keep everything on-prem.) The embeddings are stored in a **vector index**

(USearch), an efficient similarity search library. This entire flow is defined declaratively, saving you from writing boilerplate code for parsing, chunking, and indexing.
- **Built-in REST API Server:** Perhaps the biggest immediate benefit of LLM-App is that it **spins up a web server with predefined endpoints for queries** [16] [17]. The QA template's server exposes endpoints like `/v1/retrieve` (raw vector search), `/v2/answer` (ask a question about your documents), and `/v2/summarize` [18]. When you ask a question via `/v2/answer`, the service will retrieve relevant chunks from the vector store and then call an LLM (which you specify in the config) to generate a answer. This means you **don't have to write your own FastAPI app** for the PoC – the template covers it. The "SummaryQuestionAnswerer" component handles taking the user query + retrieved context and formulating a response [19]. You can configure which LLM to use (OpenAI GPT-4, GPT-3.5, or even local via Ollama or Hugging Face). In fact, LLM-App includes templates for private deployment with open models (e.g. using **Mistral-7B via Ollama** [20]).

Usage in this project: If you want a **quick Proof-of-Concept** with minimal coding, LLM-App's Pathway template is a strong option. You could: (1) Install Pathway and the `llm-app` template, (2) modify the `app.yaml` to point to your local invoice directory as the data source and configure the LLM (maybe start with GPT-3.5 via API for simplicity, then swap to local model later), and (3) run the app. Instantly, you get a running service where you can `POST /v2/answer` with a question like "What is the total on invoice #INV123?" and get an answer. The service handles reading new invoice files, parsing (via Docling) and updating the index continuously [11]. This can **grow into a SaaS** easily: Pathway pipelines are cloud-friendly (their docs show one-click deploy to AWS/GCP [21]), and you can add connectors for other sources (databases, cloud storage) as needed. The trade-off is some initial setup overhead (learning the YAML config format, installing the Pathway engine which is less widely known than LangChain). Also, using the Pathway template means working within its abstractions (which are quite flexible, but you might have to adapt if you want custom OCR like DeepSeek instead of Docling – more on that below).

DeepSeek-OCR – Next-Gen OCR with Invoice Intelligence

DeepSeek-OCR is a state-of-the-art open-source OCR model tailored for documents. It differs from traditional OCR in that it uses a vision-language approach: it **encodes entire document images into "vision tokens" and decodes them with a 3B-parameter Mixture-of-Experts model** [22] [23]. The result is extremely efficient and accurate text extraction, even for complex, long documents. For invoice processing, DeepSeek-OCR is a game changer:

- **High Accuracy on Invoices:** DeepSeek is trained on diverse financial documents and "actually understands invoice structure" [24]. It doesn't just spit out all text; it **distinguishes key fields**. For example, it can identify which number on the page is the *total* vs. the tax or subtotal [25]. It extracts vendor names, dates, line items, etc., with a layout-aware understanding. In tests, you can get a **clean JSON output of all key fields (vendor name, issue date, total amount, tax, currency, line items, etc.)** from a single invoice image/PDF [26]. This structured output saves a ton of effort – instead of manually parsing raw text or writing regexes for each vendor format, you get a normalized data schema in seconds.
- **Robust to Variations:** Because of its deep learning approach, DeepSeek handles *"messy" or varied layouts* that foil traditional template-based OCR. Whether an invoice is a crisp digital PDF, a wrinkled scan, or multi-language, DeepSeek-OCR performs consistently [27]. This aligns with a "zero-template" goal (no hardcoded formats) – crucial for scaling to many vendors.
- **Open-Source & Local-Friendly:** DeepSeek-OCR is MIT-licensed open source [28]. You can run it locally on your Mac (via PyTorch) with no API calls or fees, avoiding vendor lock-in. Despite being a

3B parameter model, its efficiency in using **"vision tokens" yields high throughput – 200k+ pages per day on a single GPU** has been reported [29]. On an M3 MacBook with an MPS-enabled PyTorch, you could potentially OCR hundreds of invoices in a few minutes. Even if not at peak speed, it's fast enough for an MVP (and far cheaper than paying per-page OCR APIs).

Usage in this project: DeepSeek-OCR would be the **OCR engine for PDFs and scanned images**. For each invoice file, you can use DeepSeek's model (available on HuggingFace [30] or GitHub) to get either text or structured JSON. The simplest integration is: feed an image/PDF, receive a JSON with fields. You can then **index both the raw text (for semantic search) and the structured data (for precise queries)**. For example, store the extracted text in the vector index (so that if a question mentions a product name or an amount, it can be found via embeddings), and store the JSON fields in a database for exact look-ups (e.g. a question like "What's the total of invoice 1001?" could be answered by directly retrieving the `total` field). Initially, you might just use the text + an LLM, but having the structured data opens up advanced possibilities (like programmatic querying or cross-invoice analysis by an agent).

DeepSeek-OCR can be used standalone or *in combination* with RAGFlow/LLM-App frameworks. For instance, IBM notes that **DeepSeek-OCR and Docling are complementary** [13] – you could use DeepSeek as a high-accuracy text extraction front-end, then feed that text into Docling or another parser to enrich formatting or table structure. In our case, since DeepSeek already gives structured JSON, we likely don't need Docling for invoices (Docling might help for preserving table formatting, but if JSON line items are present, we have what we need). If you use RAGFlow, you may integrate DeepSeek by writing a custom ingestion step (RAGFlow's orchestrable pipeline could call DeepSeek on images before embedding). With LLM-App/Pathway, you could replace the OCR stage: e.g. write a custom Pathway parser node that calls DeepSeek (instead of Docling) to parse invoices – Pathway is flexible enough for custom components [31]. However, this integration might require additional coding. As a simpler route, you could pre-process all PDFs/images through DeepSeek *offline* (e.g. a script that outputs JSON/text), and then feed those results into whichever pipeline (RAGFlow, LangChain, etc.) for indexing. The bottom line: **DeepSeek-OCR ensures your invoice ingestion is accurate and automated**, eliminating manual data entry and errors [32], which is critical for any finance AI assistant.

LangChain or LlamaIndex – Custom RAG Assembly (Alternative Approach)

Instead of using a large framework like RAGFlow or Pathway's LLM-App, you can build the RAG pipeline yourself with libraries like **LangChain** or **LlamaIndex**. These libraries provide components to chain together the steps (document loaders, text splitters, vector stores, LLM query prompts, memory, etc.) in a flexible way:

- **LangChain** is a popular Python framework that simplifies connecting an LLM with a vector store and other tools. For example, you can use a *ConversationalRetrievalChain* which will: embed the user query, retrieve relevant docs, and construct an LLM prompt with context all in one call. LangChain supports many vector databases (FAISS, Pinecone, Qdrant, etc.) and LLM APIs or local models. It also has memory modules to maintain conversation context across turns. Using LangChain, you could relatively quickly implement the pipeline: use a file loader to read invoices (possibly after OCR), use LangChain's text splitter to chunk, store embeddings in (say) FAISS, and then answer queries with a few lines of code. The appeal is **fine-grained control** in Python – if RAGFlow or LLM-App feel too "black-box" or heavy, LangChain lets you start small and customize. The trade-off is you'll need to handle things like designing the prompt template for the LLM, managing the FastAPI endpoints, etc., which frameworks like LLM-App provide out-of-the-box.

- **LlamaIndex (GPT Index)** is another library that is focused on indexing documents for LLM queries. It can be seen as an alternative to LangChain's retriever+LLM orchestration. LlamaIndex provides high-level indices (tree, list, vector) and can automatically manage embeddings and retrieval. It's quite developer-friendly for ingesting a set of documents and then querying them with natural language. In the architecture file you provided, LlamaIndex was suggested as part of the core stack [33]. It integrates well with Pandas (for Excel) and can be extended with custom readers (you could plug DeepSeek-OCR into a LlamaIndex loader). If you prefer a data-centric approach where each invoice becomes a node with metadata, LlamaIndex is worth considering.

Usage in this project: If you decide *not* to use RAGFlow or LLM-App's templated approach, then using LangChain or LlamaIndex is recommended to minimize development effort in assembling the RAG pipeline. You would manually implement the FastAPI endpoints: e.g., an `/ask` endpoint that calls your LangChain chain. Given you are comfortable with Python, this approach gives you transparency at each step (you can log which documents were retrieved, adjust the prompt, etc.). This also makes it easier to integrate the **structured data** from DeepSeek. For instance, if DeepSeek's JSON has a field "total", you might directly incorporate that into an answer for specific questions (bypassing the LLM to avoid error), or include those fields in the context so the LLM cites them. LangChain doesn't do this automatically, but you can code that logic (e.g. a simple if/else: if query matches a known pattern like "total of invoice X", pull from JSON; otherwise do normal LLM answer). This kind of customization is easier when you have code-level control.

Other Notable Tools (Vectors, Storage, Interface)

To complete the stack, a few more components are needed:

- **Vector Database:** For the PoC, an in-memory vector store like **FAISS** (by Facebook) works well for local use. It's just a library you pip-install and use to store embeddings and query by similarity. If you use Pathway (LLM-App), it uses **USearch** by Unum, which is similarly an efficient vector index (the Pathway pipeline abstracts this for you [34]). For a more persistent storage (MVP/SaaS stage), you might consider an open-source vector DB service like **Qdrant** or **Weaviate**, or even use PostgreSQL with the **pgvector** extension. In fact, an "all-in-Postgres" approach is suggested to reduce infrastructure – storing vectors in Postgres (via pgvector) means you don't need a separate Pinecone/Redis/etc., simplifying deployment [35]. For the MVP, you could run a Postgres on your Mac and use it for both application data and vector search to great effect (and it will scale reasonably well for moderate data sizes).
- **LLM Model:** The choice of LLM will affect local feasibility and answer quality. During initial prototyping, you might use OpenAI's GPT-3.5 or GPT-4 API for convenience (they generally give excellent answers with fewer tweaks). However, to stick to open-source and to eventually minimize costs, you can run a local model. Options include **Llama 2** (7B or 13B parameter variants) or newer small models (Mistral 7B, etc.). On an M3 MacBook, a 7B model can run with GPT-4-like quantization using Metal performance shaders – it's doable, though still relatively slow compared to an API call. You could use a tool like **Ollama** (which is a Mac-focused runtime for LLMs) to serve a model like Llama-2 7B and call it from your app, or load the model in PyTorch with `torch.mps` enabled. There's a trade-off between speed vs. openness: perhaps use GPT-3.5 in PoC to validate the pipeline, then switch to an optimized local model for the MVP. Notably, RAGFlow updates indicate it supports the latest open models (like Mistral and even GPT-5 series) and can integrate with local runtimes [36]. Pathway's template likewise has examples using Mistral via Ollama [20]. So both frameworks can be configured for local LLMs when you're ready.

- **FastAPI (or similar)**: If not using an existing API server (like Pathway's QASummaryRestServer), you'll create a simple FastAPI (or Flask) app to expose your system. FastAPI is recommended for its speed and built-in docs. You might have an endpoint like `POST /query` with JSON body `{"question": "...", "conversation_id": "..."}`. The server handler would retrieve relevant invoice context and return an answer. Implementing a basic version of this is straightforward, and you can gradually enhance it (add user authentication, session management for conversation memory, etc.). This layer is what will turn your local pipeline into a usable **API product** (and later can be containerized for SaaS deployment).

The table below summarizes the key tools and their roles:

Tool / Library	Role in Pipeline	Key Features and Notes
RAGFlow [1] [3]	Full RAG engine (ingestion, retrieval, LLM Q&A)	Turnkey solution with multi-format ingestion (PDF, image, Excel) [3], hybrid search [4], agent orchestration for advanced workflows. Good for scaling up with minimal custom code, but requires configuration.
Pathway LLM-App [10] [18]	Real-time RAG pipeline + API server	Ready-to-deploy templates for Q&A over docs. Continuous indexing of new files [11], uses Docling for parsing, USearch for vectors, with built-in REST endpoints [18]. Fast to get running; highly extensible via YAML.
DeepSeek-OCR [25] [26]	OCR & field extraction for invoices	Vision AI model that reads invoices and outputs structured JSON (vendor, date, total, tax, line items) [26]. High accuracy on multi-format invoices [32]. Open-source (3B model) that can run locally, replacing manual data entry.
LangChain / LlamaIndex	RAG pipeline development libraries	Provide building blocks to implement custom ingestion, vector store, and LLM query logic in code. LangChain has chains for conversational QA and many integrations; LlamaIndex offers simplified index management. Use these if not using RAGFlow/Pathway.
Vector Store (FAISS / pgvector / USearch)	Semantic search index	Stores document embeddings for similarity search. FAISS is an in-memory library (good for local dev). **pgvector** allows storing vectors in Postgres [35] (unifying data storage). USearch (used by Pathway) is a fast C++ index embedded in the app.
LLM (e.g. Llama 2, GPT-3.5)	Answer generation (with context)	Large Language Model to generate answers from retrieved context. Open-source options (Llama 2 7B/13B, Mistral 7B) can run on local hardware (with quantization). Alternatively, OpenAI GPT endpoints for quick setup. Should be interfaced through libraries or APIs.
FastAPI (or Flask)	API exposure layer	Web framework to build a conversational REST API. Handles request/response, session cookies or tokens for conversation tracking, etc. Use FastAPI to integrate with the pipeline (unless using Pathway's built-in server).

Step-by-Step Implementation Plan (PoC to MVP)

Following is a stepwise plan to implement the solution. The early steps focus on a **Proof-of-Concept (PoC)** where everything can be done on your MacBook. Subsequent steps enhance this into a more robust **MVP**, and we will then discuss scaling it to a SaaS.

1. Document Ingestion and OCR Parsing

PoC: Begin by gathering a few sample invoices in different formats (e.g. a PDF invoice, a scanned image of an invoice, an Excel or CSV invoice). Implement an ingestion script or module that processes these files and extracts text:

- **PDF/Image Invoices:** Use **DeepSeek-OCR** to parse these. DeepSeek provides a Python interface (via HuggingFace Transformers). Load the model and run inference on each file. It will output recognized text and potentially a JSON of structured fields. For PoC simplicity, you can focus on the text output. Verify that for a given invoice, you get all relevant text (item descriptions, amounts, etc.) and that important fields like total or date are correctly captured in the JSON. *Example:* for a PDF invoice, DeepSeek-OCR might output: `{ "vendor": "ACME Corp", "date": "2025-11-01", "total": "$1,234.56", "items": [ ... ] }` along with the full text of the invoice. This step **confirms OCR accuracy** and gives you data to work with.
- **Excel Invoices:** For an Excel file, you can either (a) convert it to PDF and then also feed to DeepSeek, or (b) parse it directly. A straightforward method is to use **pandas** (read Excel as DataFrame) or Python's `openpyxl`. Extract key cells (like specific columns that contain line items, total, etc.) and/or simply generate a text representation (e.g., list all rows as text lines "Item A – $100..."). Given Excel is structured, you might directly produce a JSON with fields from it as well. Ensure the format is consistent with what you do for PDF (e.g. have a "vendor", "total" etc. if available).
- **Centralize Parsed Data:** Normalize the output of the above so that each invoice is represented with: (a) a block of text (the OCR'd content or Excel content), and (b) a set of metadata (filename, perhaps invoice number or vendor as ID if you can parse it). If DeepSeek gave structured JSON, attach that too. For now, you might store this in memory (a Python dict or list). For a larger MVP, store it in a simple SQLite or PostgreSQL database so it's persistent.

MVP: As you move to MVP with more invoices, you'll want an automated pipeline for ingestion. At this stage, consider integrating with a framework: for example, use Pathway's ingestion if you chose LLM-App, or set up a periodic task (cron or background thread) to watch an "invoices" folder. Each new file dropped in triggers the OCR parse and index update (we cover indexing next). The MVP should be able to handle dozens or hundreds of files without manual intervention. If using Pathway, just point it to the folder (it handles re-indexing on file change [10]). If custom, implement a simple file watcher. Also, start thinking about error handling: e.g., if OCR fails on a file, log it and possibly fall back to a simpler OCR like Tesseract or mark the file for human review.

2. Chunking and Vectorization of Content

PoC: Once you have raw text from invoices, the next step is to prepare it for retrieval. Typically, we **split the text into chunks** (to fit vector model input limits and to allow granular retrieval). For invoices, each document might not be very long, but a single invoice could contain multiple sections (header info, line items table, totals). It can be wise to chunk by logical sections or a token length (e.g. 200 tokens per chunk). If using LangChain or LLM-App, they have utilities: LangChain's `CharacterTextSplitter` or Pathway's `TokenCountSplitter` [12] can split on sentence or token boundaries while respecting a max size. Aim for chunks that are a few sentences or one line-item each, and **include some overlap** if necessary to not lose context between chunks.

Embed each chunk into a numeric vector using an **embedding model**. For local dev, a good choice is a **SentenceTransformer** model such as `all-MiniLM-L6-v2` (very lightweight) or `multi-qa-MiniLM` – these are small (around ~100MB) and give decent semantic vectors. If you require multi-language (invoices in various languages), choose a multilingual model (e.g. `sentence-transformers/paraphrase-multilingual-MiniLM`). Alternatively, use OpenAI's `text-embedding-ada-002` via API for best quality (but that adds external dependency and cost). In Pathway's template, if you stick with it, by default it's using OpenAI embeddings [15], so you'd configure your API key; or swap in a local embedder by implementing the interface.

After embedding, **store vectors in a vector index**. For PoC, using FAISS is easiest: you can keep the index in memory. Each vector entry should be associated with metadata: at least an ID or reference to which invoice (and which chunk) it came from. This way, when you retrieve, you can fetch the chunk text and know which document it's from. If using Pathway, this is handled by USearch + Pathway's dataflows – you get an ID for each piece [34]. With a custom approach, set up FAISS or even a simple Python list of vectors (for small scale) and use cosine similarity manually. Aim to be able to **query the index**: e.g., test with a sample question "What is the total amount for ACME Corp invoice?" – convert that to an embedding (using the same model), find nearest neighbor chunks, and see if those chunks indeed contain the answer (e.g. a chunk with "Total: $1,234.56"). This validates your embeddings are working. If results seem off, you might need to fine-tune chunking or use a better embedding model (invoices might have numeric-heavy content; some embedding models might not capture numbers well – if so, consider concatenating words like "total" with numbers to give context, or use an embedding that was trained on finance text).

MVP: For the MVP, consider moving to a more robust vector store for persistence and scale. Two approaches: 1. **Use a lightweight DB with vector support** – as mentioned, **PostgreSQL + pgvector** is an excellent choice to keep things simple (it avoids introducing a separate DB product) [35]. You'd load your embeddings into a Postgres table with a vector column. This also allows you to store the JSON fields alongside, in the same record, which is convenient for later filtering (e.g. you could do SQL like "SELECT ... WHERE metadata->>vendor = 'ACME Corp' ORDER BY embedding <-> query_embedding LIMIT 5" to combine vector search with a vendor filter). In the SaaS context, Postgres can also handle user-specific data separation (e.g. include a tenant ID column). 2. **Use an open-source vector DB** – If you anticipate very large scale or want advanced features, you might use Qdrant or Weaviate. These can run as a service (locally via Docker or managed in cloud) and offer fast approximate nearest neighbor search, plus filtering. For MVP scale (hundreds or thousands of invoices), this might be overkill – pgvector or even FAISS might suffice.

Also, as you ingest more documents, you might incorporate **hybrid search**: storing a traditional full-text index or at least the raw text in a search-friendly way. RAGFlow for instance does this automatically (combining BM25 with vectors) [4]. In a custom stack, you could use Whoosh or Elasticsearch for keyword search in parallel with vector search. Hybrid search can help if a user query is very specific (like an exact invoice number or a rare term) that maybe wasn't well-captured by embeddings. This is a nice-to-have for MVP if time permits.

3. LLM Integration for Q&A

With documents indexed and retrievable by similarity, the core "knowledge" base is ready. Now integrate the **LLM** to turn retrievals into conversational answers:

PoC: Implement a query workflow: 1. **Query Understanding**: When a question comes in (text), you may do a little pre-processing. For instance, you might want to extract any explicit reference like invoice numbers or vendor names. This could be done with regex or a simple named entity check. *Why?* Because if the user says "What's the total on invoice 1005?", it's efficient to directly look up invoice `1005` in your structured data if available, rather than rely on semantic search. For PoC, this step is optional – you can just treat the query as a whole for semantic search. 2. **Retrieval**: Embed the user's question using the same embedding model and perform a similarity search in the vector index. Retrieve the top *k* chunks (commonly k=3-5) that are most relevant. These chunks (text snippets from invoices) are the context you'll feed to the LLM. If your pipeline stored metadata like document name or page, you might want to group or filter results to avoid redundancy (e.g. if top 3 results all came from the same invoice, that's fine, it means that invoice is very relevant). 3. **Prompt Construction**: Construct a prompt for the LLM that includes the retrieved context and the question. A common pattern is a system/message prompt that instructs the LLM to answer using the provided document text and not to make up information. For example:

~~~
```
SYSTEM: You are a financial assistant with access to invoice data. Answer the
question truthfully using the provided document excerpts. If you don't find the
answer, say you don't have that information.
USER: Question: "What is the total amount on invoice 1005?"

Document Excerpts:
1. Invoice 1005 - ACME Corp ... Total: $1,234.56 ...
2. Invoice 1005 - ... (more context) ...
QUESTION: What is the total on invoice 1005?
```
~~~

Tailor the template as needed. If you plan to output sources or structured answers, instruct accordingly. For a conversational tone, you might let the assistant answer in a friendly manner, but given this is financial data, accuracy is paramount – emphasize citing the context or being sure. If using LangChain, it can handle some of this prompt logic for you (the `ConversationalRetrievalChain` will integrate the context into a prompt template you provide). If using Pathway's QASummary, it likely has an internal prompt that does summarization or QA; you can adjust it in the config.

1. **Call LLM**: Use the chosen LLM to generate an answer. For PoC, easiest is an API call to GPT-3.5 (via `openai` Python package) – ensure to pass the prompt with the context. Check that the answer looks reasonable and factual given the context. If you're using a local model (e.g. via HuggingFace pipeline), load it and generate similarly. You might need to experiment with smaller models to see if they can answer accurately with the context given. Some 7B models might struggle with arithmetic or multi-step reasoning (e.g. if the question is "What's the total of all invoices from ACME in September?" – that requires summing multiple documents). For now, keep questions simple (one

invoice at a time). Complex queries can be handled later with agentic strategies (see upgrade section).

2. **Return Answer**: For PoC, just print the answer or return from a function. This whole flow can be encapsulated in a function `answer_query(question)` that the API will call. Test it with a few example questions to ensure it retrieves the right context and the LLM responds sensibly. Evaluate edge cases: ask something not in any invoice ("What's the CEO's name?") – the system should ideally respond it doesn't know, rather than hallucinate. Adjust the prompt or retrieval as needed to reduce hallucinations (RAG already helps; you can also configure the LLM to be more conservative).

MVP: In the MVP stage, you will integrate this into the API server (so the answer generation happens per request). Additional enhancements for the LLM component could include: - **Conversational Memory:** If you want multi-turn dialogue (where the user can ask follow-ups like "What about the previous month's total?" referring to a past query), you need to maintain context of conversation. The simplest way: store the last N questions and answers for a session ID, and prepend them to the prompt on each new query (or use LangChain's memory classes). For MVP, supporting at least short follow-ups is nice. Given invoice queries are often independent, you might not need deep conversation memory, but it's good for user experience (especially if a user asks a series of related questions on one vendor). Be mindful of token limits – don't let the history become too large. - **Refine Prompting:** Based on testing, you might add instructions like "If the question asks for a list, provide it as bullet points" or "Always double-check calculations using the data." You could even have the LLM output JSON if you plan the front-end to consume structured answers (not required for API-only, but could be useful). - **Local LLM deployment:** If transitioning off OpenAI API, set up the local model fully. Possibly use the **transformers library with** `AutoModelForCausalLM` to load Llama 2 (quantized to 4-bit using bitsandbytes or so for memory). Or spin up **Ollama** with a model and call it via HTTP (Ollama is very Mac-friendly). Ensure the latency is acceptable (you may need to use a smaller model or run on CPU multithread if GPU is a bottleneck). You can also explore newer 2025 open models (there might be optimized 13B models that approach GPT-3.5 performance). - **Testing:** Develop unit tests for the query pipeline (e.g. feed a known invoice and ask "what is the total?", assert the answer contains that number). Also test the entire API call using FastAPI's TestClient or similar.

4. Conversational API Development

Now, wrap the functionality into an API service:

PoC: You can start with a basic FastAPI app with one POST endpoint. For example:

```
from fastapi import FastAPI
app = FastAPI()

@app.post("/query")
def query_invoices(request: QueryRequest):
    question = request.question
    session_id = request.session_id
    # (Optional) retrieve conversation history via session_id
    answer = answer_query(question, session_id)
    return {"answer": answer}
```

Here `answer_query` is a function you implement (from step 3) that handles retrieval + LLM. If you want to maintain sessions, you might keep a global dict of session->history, and update it with each Q&A (or use FastAPI's dependency system to manage state). Initially, you can ignore session and treat each query independently.

Test this endpoint locally (FastAPI's docs and automatic Swagger UI are handy). You can use `curl` or a REST client to send a JSON with a question and see the response. At this stage, the API simply returns the answer text. You might also include some meta-info like which invoice or confidence, but it's not strictly necessary. Keep it simple (the client asking the question likely just needs the answer).

MVP: Expand the API capabilities: - **Session/Conversation Handling:** Introduce an ID (or use something like a JWT token or API key) to identify users or sessions. The API could then isolate conversations per user. This is crucial when moving to SaaS (each user's data and queries should be isolated for security). For example, you might require an API key with each request, and that key is tied to a set of invoice data. In MVP, you can simulate this by a config file mapping an API key to a directory of invoices, or by simple multi-tenancy in the vector store (store user_id in vector metadata and filter on retrieval). LangChain's filters or a SQL where clause (if using pgvector) can implement this.
- **Additional Endpoints:** Possibly add a `/ingest` endpoint to upload a new invoice via API. This would allow external systems to send in an invoice file (PDF/image) to add to the knowledge base. The endpoint handler would run the OCR and indexing on-the-fly. This is useful for SaaS where invoices come in real-time rather than being batch loaded. You'd likely queue the file for background processing to avoid slowing the query response. Tools like Celery or just threading could be used – but be mindful on Mac to keep it simple. Pathway's pipeline, if running, would catch new files automatically, so an API might just save the file to the watched folder. - **Logging & Monitoring:** Add basic logging for each query – what was asked, which documents were retrieved, etc. This will be invaluable for debugging when a user says "I got a wrong answer on this query." It's also good to track response times (to plan scaling needs). In a SaaS, you'd also want to monitor for any errors (e.g. OCR exceptions, LLM timeouts). - **Security:** For MVP, at least include an API secret or require HTTPS if deployed. Since it's internal for now, this can be minimal, but before SaaS, you'll implement a proper auth (token-based auth for client apps, and per-user data segregation). - **Documentation:** Thanks to FastAPI, you get docs, but ensure you document expected request/response. This will matter when you expose it to third-party clients or even if you build a small UI on top.

5. Testing the PoC

Before moving to scale or SaaS considerations, validate the PoC thoroughly: - **Functional testing:** Does it answer various question types correctly? e.g. *"What is the total amount of invoice #1001?"*, *"On what date was ACME's last invoice?"*, *"Which items did John Doe purchase in his invoice?"*. For each, verify against the source invoice. If any answer is wrong or "hallucinated," trace why – did retrieval miss the relevant text (tweak embeddings or add keyword search), or did the LLM mis-read the context (maybe add more explicit prompt directions or ensure the raw text contains the needed clue). - **Performance check:** Measure how long a single query takes end-to-end. On Mac M3, DeepSeek OCR might take a couple of seconds per page (if run at query time). But note, in our design, OCR is done at ingestion, not at query time, so queries should be quite fast (the main cost is the LLM call – GPT-3.5 might be ~1-2 seconds, local LLM could be more). For now, it's fine; later, we ensure scaling can handle concurrent queries. - **Edge cases:** Try queries that reference multiple documents: *"What's the total of all ACME Corp invoices?"* – currently, the system might retrieve chunks from all ACME invoices and pass them to LLM. GPT-4 could sum them up correctly, but smaller models might not. This hints at future enhancement (an agent that can do a sum, or pre-computed analytics). Acknowledge that complex analytical queries might be out-of-scope for the initial MVP's capabilities, but note them for future.

Once the PoC is stable and answers basic questions well on your sample data, you can proceed to refine it into an MVP (which mainly means making it robust, user-friendly, and preparing for deployment).

Upgrade Path: Toward a Scalable SaaS

Transforming the MVP into a multi-user, scalable SaaS product will involve architectural adjustments, additional components for robustness, and DevOps work. Here's an outline of the upgrade path:

- **Multi-Tenancy and Data Isolation:** In a SaaS, each business (or user) will have its own invoices which must be kept separate. You'll need a way to segregate data per tenant. If you continue with a single vector DB, utilize metadata filters (like a tenant ID on each vector) to ensure queries only retrieve that tenant's docs. Alternatively, maintain separate indices per tenant (if using something like Qdrant, you can have collections per tenant; with Postgres, include tenant_id in queries). The API should enforce authentication (e.g. JWT tokens that carry tenant info) and use it to route queries to the correct data. This is a crucial security step.
- **Scalable Storage and Retrieval:** As data grows (say you onboard dozens of companies, each with thousands of invoices), the storage layer must handle it. Move from local storage to cloud:
- Document files -> store in an object storage like **S3 or MinIO** (MinIO if you want open-source S3-compatible storage) [37] . Ingestion pipeline would then fetch files from there. This decouples file handling from the app server (which is important for scale and statelessness).
- Vector store -> if using Postgres, consider a managed Postgres or Aurora for reliability, and ensure it's tuned for vector queries (pgvector can do IVF indexes for large scale). If using a specialized vector DB, you might run it as a separate service (with replication for HA).
- Standard DB -> likely you'll also have a relational DB for things like user accounts, invoice metadata, logs, etc. This could be the same Postgres (reusing it for both vectors and relational data is convenient for MVP) [35] . For scale, ensure proper indexing on any fields used for filtering (e.g. vendor name if you allow queries scoped by vendor).
- **Containerization and Deployment:** Containerize the application (Docker). Ensure that the OCR model (DeepSeek) and the LLM (if local) are packaged or accessible. You might split services: e.g., one service for the API & LLM queries, another for ingestion & OCR (especially if OCR uses GPU – you might want separate workers for that to not block queries). Using a queue (like Redis or even Postgres NOTIFY/LISTEN or **pgqueuer** as hinted in your architecture file [35]) to handle new file processing can make the system more resilient. For example, a user uploads an invoice -> it's stored in S3 -> a message is placed on a queue -> an ingestion worker VM picks it, runs DeepSeek, updates the DB. Meanwhile the query API can stay responsive.
- **Scalability Considerations:** For handling many simultaneous users/queries, you can scale horizontally:
- Run multiple instances of the API service behind a load balancer. They can be stateless (if all state is in the DB/vector store and object storage). Just ensure each instance can access the same indices. With Postgres or a centralized vector DB, that's fine. If you used an in-memory index like FAISS in-

process, you'd need to move to a centralized store for multiple instances (another reason to use a DB or external vector service).

- The LLM itself could become a bottleneck if using a local model. You might switch to a model serving solution that supports concurrency and batching (for example, **vLLM** or HuggingFace's text-generation-inference server). These allow multiple requests to share a GPU and improve throughput [38] [39]. In a SaaS, you might opt to fine-tune a smaller model on invoice Q&A to get good performance and run it on a GPU instance in the cloud. Alternatively, you can integrate with hosted LLM APIs (OpenAI, Anthropic) with proper rate limiting and cost considerations – some SaaS choose a hybrid approach (open-source for cheap tasks, call paid API for complex queries or as fallback).
- The OCR (DeepSeek) is heavy but usually only done on document upload, not per query. Still, if many docs are uploaded at once, ensure the OCR workers can scale (maybe use Kubernetes HPA to spin up more pods, or queue jobs and process sequentially if throughput is acceptable).
- **Advanced Features:** With basic Q&A working, you can add features to differentiate your product:
- **Analytics and Multi-Document Reasoning:** Users might want questions like "Total spending by vendor X last month" or "What's the average invoice amount this quarter?". These require aggregating data across documents. A simple approach is to pre-compute analytics (since you have structured data from DeepSeek, you can sum totals per vendor, etc., in your DB and let the LLM or API fetch those numbers). A more dynamic approach is to incorporate an **Agent** that can use tools: e.g., use an LLM that can call a calculation function or SQL. RAGFlow's agent orchestration could shine here – you could configure an agent with a tool that queries the database directly for sums. LangChain also has tool use functionality (like giving the LLM a Python REPL or SQL query ability). This moves beyond MVP, but it's a natural progression to answer analytical questions reliably (the LLM essentially becomes a natural language interface to a database of invoice data).
- **Validation & Self-correction:** Invoices are sensitive to errors. You may implement validation agents (as in your file's "Validation Agent" concept) that verify the extracted data makes sense (e.g. subtotal + tax = total) [40] [41]. This could be done at ingestion: after DeepSeek, have a routine or even an LLM double-check calculations and flag anomalies. This ensures the knowledge base isn't garbage-in-garbage-out. RAGFlow or an agent could be configured to attempt alternate OCR or request human verification if something looks off. For SaaS, building a "human-in-the-loop" interface (maybe a simple Streamlit or web dashboard for admins to correct a mis-read field) can be crucial in industries like finance for auditability.
- **Conversational UX improvements:** Consider integrating the API with a chat frontend (even though you don't need a full dashboard, having a simple web chat for demos is useful). There are tools like **Vercel's AI chat UI** or frameworks like **Streamlit** that can quickly create a chat interface for users to converse with the system. This can later be embedded into a product (e.g. a chat widget on a finance system). Since your API is stateless (assuming you pass a session ID for context), it can serve as the backend for any front-end.
- **Monitoring and Observability:** As a SaaS, you'll need to monitor usage (for billing, or to watch for abuse). Integrate logging that tracks number of documents processed per user, number of queries, response latency, etc. Open-source observability tools (like Grafana with Loki/Promtail for logs, or LangChain's tracing if using it) can be set up. This will help you pinpoint performance issues and also provide metrics to show value (e.g. "our AI answered 500 questions and saved X hours of manual lookup").
- **Extensibility:** Keep the system modular so new features can plug in. For instance, maybe tomorrow you want to integrate with email (auto-read invoices from an email inbox), or add support for other document types (purchase orders, receipts). If you've built on a solid open-source stack, you can extend ingestion connectors and adjust parsing relatively easily. The frameworks chosen (Pathway, RAGFlow, LangChain) all emphasize extensibility – e.g., RAGFlow can sync with Confluence, Google

Drive, etc., via built-in connectors [7] which you can leverage to pull in invoices from various sources in future.

Tech Stack Recap for SaaS: You will likely move from a monolithic MacBook app to a cloud environment with **multiple containers/services**. For example: - **Web App** (FastAPI) – runs the API, does retrieval, calls LLM (this could be scaled out to N replicas). - **LLM Model Server** – if using a heavy model, you might separate it (e.g. run a dedicated GPU pod with the model and have the API pods call it). Or use a managed LLM API. - **Ingestion Worker** – a service that listens for new files (from an upload service or bucket events) and runs DeepSeek-OCR + indexing. This can be on a GPU instance if needed. - **Databases** – Postgres for metadata and vectors (with replicas or a cluster if high load), and an object storage for files (S3). - **Cache** – you might introduce a cache like Redis if needed (for example caching LLM answers to identical questions to save compute, or storing session conversation state if you don't want to pass it every time). - **CI/CD and Infrastructure-as-Code** – as you productize, set up Docker builds, testing pipelines, and use IaC (Terraform or similar) to define cloud resources. Leverage container orchestration (Kubernetes or simpler ECS) for deploying the pieces. Both RAGFlow and Pathway provide Docker images which you could adapt [42], and they have documentation for deployment which can accelerate this phase.

Finally, ensure that going to SaaS doesn't sacrifice the **cost advantage**. By using open-source models (DeepSeek for OCR, local LLMs) and unified infrastructure (Postgres for multiple purposes) [35], you significantly cut ongoing costs compared to relying on cloud APIs for OCR or vector DB. Continuously profile the system to keep it efficient – e.g., if certain queries are slow, see if you need better indexes or more RAM. The "Complexity Collapse" approach in your notes – using a cost-effective unified stack – is exactly what we're following with this open-source RAG architecture.

Conclusion

In summary, the recommended solution is to **combine the strengths of open-source RAG frameworks with a powerful OCR model and a lean development approach** to rapidly build an invoice Q&A chatbot. Tools like RAGFlow or Pathway's LLM-App can jump-start the development by handling ingestion, indexing, and even serving endpoints, whereas DeepSeek-OCR ensures high-quality data extraction from invoices (the foundation for correct answers). For a developer on Mac/Python, this means you can get a prototype working quickly, then iteratively replace any piece with your preferred implementation (e.g. switch out the embed model or LLM, add custom logic for certain queries) – the system is modular.

By following the steps from ingestion to API deployment, you will create a functional PoC that answers questions like *"What's the total on this invoice?"* or *"List the line items on Jane's order"* accurately by referencing the actual invoice content. From there, you can harden and expand the system into a SaaS offering, handling more users and documents, and providing natural, conversational access to financial data. With this approach, you leverage open-source innovations (like DeepSeek's vision tokens and RAGFlow's context fusion) to deliver a powerful solution without incurring the heavy costs of proprietary services – exactly the ethos of **minimizing dev effort and cloud expense while maintaining extensibility**.

---

[1] [3] [5] [6] [7] [8] [36] [42] GitHub - infiniflow/ragflow: RAGFlow is a leading open-source Retrieval-Augmented Generation (RAG) engine that fuses cutting-edge RAG with Agent capabilities to create a superior context layer for LLMs
https://github.com/infiniflow/ragflow

[2] [4] RAGFlow
https://ragflow.io/

[9] Top 10 RAG Frameworks Github Repos 2025 | by Rowan Blackwoon
https://rowanblackwoon.medium.com/top-10-rag-frameworks-github-repos-2025-dba899ae0355

[10] [11] [12] [14] [15] [16] [17] [18] [19] [20] [21] [31] [34] Question-Answering RAG App | Pathway
https://pathway.com/developers/templates/rag/demo-question-answering/

[13] [23] [29] DeepSeek’s new open-source model could decode large amounts of documents more efficiently | IBM
https://www.ibm.com/think/news/deepseeks-new-model-could-decode-documents-more-efficiently

[22] deepseek-ai/DeepSeek-OCR: Contexts Optical Compression - GitHub
https://github.com/deepseek-ai/DeepSeek-OCR

[24] [25] [26] [27] [30] [32] DeepSeek-OCR in Invoice Processing Automating Finance Workflows - Skywork ai
https://skywork.ai/blog/llm/deepseek-ocr-in-invoice-processing-automating-finance-workflows/

[28] How DeepSeek OCR Quietly Solved a Billion-Dollar Problem in AI ...
https://medium.com/data-and-beyond/how-deepseek-ocr-quietly-solved-a-billion-dollar-problem-in-ai-scaling-7b4502613af9

[33] [35] [37] [40] [41] README.md
file://file_0000000018b4720cae375290dfba9386

[38] [39] GitHub - garylab/MakeMoneyWithAI: A list of open-source AI projects you can use to generate income easily.
https://github.com/garylab/MakeMoneyWithAI