ChatGPT

AI System Architecture Discussion

FastAPI Backend: RAG Ingestion vs. Chat Orchestration

A FastAPI service is well-suited for handling document ingestion, embeddings, and basic RAG (Retrieval-Augmented Generation) queries. In fact, many RAG pipelines use FastAPI to expose endpoints for uploading documents and querying them. For example, one reference implementation uses FastAPI to expose **"ingestion and query endpoints"** for PDF documents, running embeddings and a local Chroma vector store behind the scenes [1]. This means FastAPI can handle uploading PDFs, chunking them, calling OpenAI (or similar) to embed those chunks, storing vectors, and then serving a query endpoint that performs similarity search and LLM completion to answer questions grounded on the PDF content [1] [2].

The question is whether FastAPI should *also* manage the conversational logic (session state, turn-taking, chat history, etc.), or whether to offload that to an agent framework (like OpenAI's Agents SDK on Cloudflare Workers). Both approaches are feasible:

- **FastAPI as RAG backend only:** In this model, FastAPI focuses on data services (embeddings, vector store, PDF ingestion) and possibly a simple `/query` endpoint that takes a question and returns an answer (with references). The frontend (e.g. a Cloudflare Workers + Agents app) handles the chat flow, history, and any multi-step planning. This decouples the chat orchestration from the data layer. For example, one tutorial demonstrates a FastAPI service that provides a `/query` endpoint for an AI to answer questions using PDF documents [1], while the conversational agent (running elsewhere) manages dialog.

- **FastAPI as full chat orchestrator:** Alternatively, FastAPI can host the entire chat agent logic (using a library like LangChain or similar). In this setup, you might create a LangChain agent (with tools and memory) on the Python side, then serve it via a FastAPI endpoint (e.g. `/ask`). An example article builds a LangChain-based agent (with memory and external tool calls) and exposes it through a FastAPI `/ask` API [3] [4]. This demonstrates that all conversational reasoning can happen in Python: the agent fetches any needed data (via APIs or vector searches) and the `/ask` endpoint simply forwards user input to that agent.

- **Using OpenAI Agents SDK on Workers:** A middle ground is to use FastAPI for the data layer (ingest, embeddings, query) and let a TypeScript-based agent framework handle the chat. OpenAI's new Agents SDK (for JavaScript/TypeScript on Cloudflare Workers) is explicitly designed for conversational orchestration. For instance, a tutorial on building an "agentic" FastAPI backend notes that you can bolt an agent on top of your normal API to make it conversational [5]. The quote says the Agents SDK adds "some magic" to make a backend fully chat-capable [6]. In that pattern, you might have the Workers/TS agent call your FastAPI APIs as tools (for example, a `/query` endpoint) and manage the dialogue and memory on the client/edge side.

Recommendation: If your team already plans to use the OpenAI Agents SDK in Cloudflare Workers, it makes sense to keep FastAPI focused on heavy lifting (PDF ingestion, embedding storage, RAG retrieval) and let the Agents SDK manage turn-taking and complex conversation logic. This leverages each layer for its strengths. However, if you prefer a unified Python backend or simpler architecture, FastAPI +

LangChain can handle chat orchestration too [3] [4]. Both are viable; using the Agents SDK could simplify the frontend/chat logic at the cost of splitting responsibilities, whereas doing everything in FastAPI centralizes the logic but can be more custom to implement.

Vector Store Options: Cloudflare Vectorize vs. Pinecone (vs. KV)

For storing and querying embeddings, Cloudflare's **Vectorize** offers a compelling option if you're already on Workers. Vectorize is a fully managed, globally distributed vector database integrated with the Cloudflare platform [7]. It lets you store high-dimensional embeddings and perform fast nearest-neighbor searches with low latency. Importantly, Vectorize is built to work seamlessly with other Cloudflare services: for example, results from a Vectorize query can directly reference documents in R2 or KV without leaving the Workers environment [8]. This means you can keep all data (vectors, documents, metadata) on the Workers platform, avoiding extra infrastructure. Given this tight integration, Vectorize is a natural default choice for a Workers-based app: it's "faster, easier and more affordable" to query embeddings [7] and it offloads the complexity of running your own vector index.

Pinecone is a mature alternative. Pinecone is a standalone, fully managed vector database (hosted by Pinecone) known for high scalability, low-latency search, and robust feature set [9]. It supports real-time ingestion of vectors and has SDKs in Python/JS (and even integration with LangChain [10]). Unlike Vectorize, Pinecone is not tied to Cloudflare – it runs in its own cloud environment. That means if you use Pinecone, your FastAPI or Workers code would make external API calls to Pinecone's service. Pinecone's advantages are its proven scalability and track record (it was even named an AI Innovator by Fortune [11]). The downside is an extra external dependency and potential latency cost compared to an edge-integrated DB. Still, Pinecone is "purpose-built to tackle the challenges of high-dimensional data" [9] and can be a reliable fallback if Vectorize doesn't meet requirements.

Using **KV (Workers KV)** as a vector store is generally not recommended for large-scale retrieval. Workers KV is a simple key-value store (eventually consistent and optimized for small reads/writes), not a vector search engine. It can be used for caching or indexing tiny datasets, but it cannot perform ANN (Approximate Nearest Neighbor) searches on embeddings. In theory, one could store embeddings in KV and scan them manually, but this would be very slow and unscalable. Therefore, KV might serve as a last-resort small fallback (e.g. store a handful of critical vectors), but for any real vector search we recommend using Vectorize or Pinecone.

Summary: If staying within Cloudflare Workers, **Vectorize** is the natural choice for embedding storage [7] [8]. If you need an external option or are already using Pinecone, Pinecone is a solid, scalable solution [9]. Using Workers KV as the *primary* vector store is not ideal due to its lack of vector-search capabilities.

Authentication: Google-Only Login and Supabase

For an auth system supporting only Google login, **Supabase Auth** is a convenient solution. Supabase Auth has built-in social sign-in providers, including Google OAuth [12]. It easily allows you to enable *only* the Google provider (and disable other sign-in methods), so users can log in with their Google accounts and nothing else. Setting this up involves minimal work: Supabase's dashboard lets you configure Google client ID/secret and define allowed domains. The documentation explicitly mentions Google sign-in support for web and mobile apps [12], indicating it's a first-class feature.

Using Supabase has the additional benefit that it integrates well with other Supabase features (like its database and storage) if you're already using them. If you don't need a full database solution, you could also consider other OAuth providers or libraries (for example, Firebase Auth or third-party services like Auth0 or Clerk), but these would also support Google and require similar setup. Given the question specifically suggests Supabase, and assuming you like its ecosystem, Supabase Auth is a sensible choice for Google-only SSO.

One thing to note: if you truly want *only* Google logins, make sure to disable any default email/ password or other social providers in your auth config. Supabase lets you do that so that the only enabled method is Google. The user experience then is simply “Sign in with Google” and no other option.

Data Sources: Public APIs + RAG Fallback

Combining structured API data with an unstructured RAG fallback is a common design pattern. Here’s how it could work: first try to answer a query using a reliable public API (or database) for horse-related data (e.g. stats, records, race results). If the API returns a definitive answer, use it. If the API doesn’t cover a particular question (or the answer is incomplete), then fall back to your document/PDF RAG system. In the RAG mode, you query the vector store of horse-related documents (rules, breed information, manuals, research papers, etc.) to retrieve relevant text snippets, then feed those into the LLM for an answer.

Retrieval-Augmented Generation (RAG) is exactly this concept: fetch relevant “chunks of knowledge” and have the LLM answer with that context [2]. In practice, one might orchestrate this in an agent or code: attempt API call(s) first, and only if needed, perform the vector search over PDFs. This hybrid approach ensures answers are grounded in reliable data when possible, with the flexibility of documents for edge cases. Many AI Q&A systems do this; for example, Cloudflare’s AI Search (AutoRAG) service explicitly uses RAG to augment answers from indexed data [13].

In summary, yes—blending public APIs with an AI/RAG fallback is reasonable. Your system can prioritize factual, structured data from APIs and then use the PDF-based RAG as a supplementary knowledge source. This maximizes correctness (use trusted APIs) while still giving the model the ability to answer deeper or unusual questions via document search. Just ensure your architecture can route queries through both pathways (for instance, the agent or chat logic decides when to call the vector-query endpoint).

Sources: FastAPI backends are commonly used for PDF ingestion and RAG APIs [1]. OpenAI’s Agents SDK can sit on top of such a service to enable conversational chat [5]. Cloudflare Vectorize is explicitly designed for embedding search on Workers [7] [8], while Pinecone is a proven external vector DB [9]. Supabase Auth supports Google sign-in out of the box [12]. And retrieval-augmented generation (“RAG”) – fetching document chunks before answering – is a well-known pattern for handling unstructured data [2] [13].

---

[1] [2] Building a PDF-Powered RAG Service with FastAPI, Chroma, and OpenAI | by Brilian Ade Putra | Oct, 2025 | Medium
https://medium.com/@brilianadeputra/building-a-pdf-powered-rag-service-with-fastapi-chroma-and-openai-1ca773e94eb3

[3] [4] Building an Agentic AI with LangChain, Azure OpenAI, and FastAPI | by Pritam Paul | Sep, 2025 | Medium
https://medium.com/@pablopaul1999/building-an-agentic-ai-with-langchain-azure-openai-and-fastapi-b570f14d9106

5 6 Building a Simple Agentic Backend with FastAPI, Supabase, and the OpenAI Agents SDK | by 404foundme | Aug, 2025 | Medium
https://medium.com/@404foundme/building-a-simple-agentic-backend-with-fastapi-supabase-and-the-openai-agents-sdk-a93fc1ce21bf

7 8 Overview · Cloudflare Vectorize docs
https://developers.cloudflare.com/vectorize/

9 10 11 The 7 Best Vector Databases in 2025 | DataCamp
https://www.datacamp.com/blog/the-top-5-vector-databases

12 Login with Google | Supabase Docs
https://supabase.com/docs/guides/auth/social-login/auth-google

13 Cloudflare AI Search · Cloudflare AI Search docs
https://developers.cloudflare.com/ai-search/