1. Fluxo (passo a passo prático)

2. Cliente envia: `message` + `userProfile` + `files[]` (multipart).
3. Backend valida arquivos (tipo, tamanho, scan) → armazena temporariamente (S3 ou pasta `uploads/` fora do repo).
4. Para cada arquivo:
   a. Extrai texto (PDF/DOCX/PPTX/TXT — e OCR para imagens/PDF escaneado).
   b. Gera metadados: pages, headings extraídos, tamanho, autor (se houver).
5. Chunking do texto em pedaços com overlap.
6. Gerar embeddings por chunk (OpenAI ou outro).
7. Indexar chunks em vectorDB (Supabase).
8. Rodar RAG por documento:
   a. Extrair **TOC** (sumário automático).
   b. Extrair **lista de tópicos** (cada tópico: título curto + descrição 1–2 frases).
   c. Para cada tópico, retornar **exemplos de trechos** (chunk ids).
9. Obter estrutura-base do curso (pipeline atual).
10. **Merge / Match**:
    a. **Strong match (≥ 0.75):** não criar novo tópico — apenas **vincular** o tópico do documento ao tópico já existente. Esses trechos vinculados serão usados depois (quando usuário aprovar) como contexto para gerar a aula-texto e as buscas no YouTube.
    b. **Weak match (0.60–0.75):** tópico da estrutura cobre **parte** do que o documento traz — **quebrar / granularizar** o conteúdo do documento (ou o tópico do curso) para descobrir partes não cobertas e criar sub-tópicos/novos tópicos apenas para as lacunas.
    c. **No match (< 0.60):** criar novo tópico (obrigatório: todos os tópicos do documento devem aparecer no curso final).
11. Para tópicos da estrutura-base sem correspondência em documentos:
    a. Marcar como `no_doc_match`. Se usuário preferir, pedir ao GPT para gerar "ponte" (resumo/trecho) com base em web/perplexity ou sugerir leitura.
12. Gerar estrutura final JSON: módulos, tópicos, sub-tópicos, para cada tópico a lista de `documentMatches` (file, chunkId, score, excerpt).
13. Dados dos documentos enviados devem ser salvos no banco para poder ser usados como contexto para criação do curso depois, esse contexto pode ser usado para criar a aula texto do topico que é ligado ao documento, e filtro de busca do video do youtube

Importante:

- **Todos os tópicos vindos dos documentos devem aparecer** no curso final (crie novos tópicos se necessário).
- Não duplicar títulos idênticos

2) Melhorias recomendadas (prioridade prática)

1. **Pipeline com filas e workers**
    a. Upload síncrono só valida/guarda arquivo. Extração, OCR, chunking e embeddings rodar **em workers** via fila (Redis + BullMQ, ou SQS + Lambda). Isso evita timeouts e melhora escalabilidade.
2. **OCR layout-aware** para documentos complexos
    a. Use AWS Textract / Google Document AI / Azure Form Recognizer para PDFs complexos (layout, tabelas) — Tesseract só quando for algo simples.
3. **Conversão padronizada**
    a. Sempre converter arquivos para um formato intermediário (plain text + structural metadata: headings/pages/blocks). Use PyMuPDF/pdfplumber para PDF, python-docx para DOCX, python-pptx para PPTX.
4. **Token-based chunking** (não caracteres)
    a. Fazer chunk por tokens (por ex. ~800–1200 tokens por chunk, overlap 150–300 tokens) usando o tokenizer do modelo de embedding. Evita cortes no meio de frases/expressões.
5. **Normalização e canonicalization de títulos**
    a. Normalizar titles (lowercase, strip diacritics, remove punctuation), indexar um hash para evitar duplicatas; usar fuzzy-match / Levenshtein antes de criar novo tópico.
6. **Dual-embedding / fallback**
    a. Se embedders (OpenAI) tiverem falhas, mantenha fallback local (ex.: text-embedding-3-small / outro provedor) e marque versão do embedding no metadado.
7. **Confidence calibration & AB test**
    a. Calibre os thresholds com um set de teste — registre TP/FP para ajustar 0.75/0.60 conforme idioma/vertical.
8. **Granularização automática por clustering**

a. Para weak match (0.60–0.75), clusterize os chunks relacionados e proponha sub-tópicos automaticamente (usar HDBSCAN ou k-means em embedding space).

3) Ferramentas / serviços / libs recomendados (concretos)

Infra / serviços

- Object Storage: **AWS S3** (ou DigitalOcean Spaces).
- Queue / Workers: **Redis + BullMQ** (fácil com Node/TS) ou **AWS SQS + Lambda / ECS**.
- Vector DB: **Supabase (pgvector)** — já no seu stack; alternativas: **Pinecone**, **Weaviate**, **Milvus**.
- Embeddings: **OpenAI text-embedding-3-large** (principal). Mantenha opção de fallback (ex.: Cohere, Hugging Face).
- OCR / Document AI: **AWS Textract** ou **Google Document AI** (preferível para layout/tabelas). Tesseract para fallback offline.
- File scanning (segurança): **ClamAV** em pipeline, ou serviço de sandboxing (VirusTotal API) para uploads suspeitos.
- Conversão/Parsing: Python libs — **PyMuPDF (fitz)**, **pdfplumber**, **pdfminer.six**, **python-docx**, **python-pptx**. Para conversões complexas: LibreOffice headless / unoconv.
- Backend / Hosting: Next.js API routes + workers (ECS/Fargate or vercel + background queue via separate worker).
- Observability: **Sentry**, **Prometheus/Grafana**, logs estruturados (elk).

Libs / frameworks

- Orquestração RAG / retrieval helpers: **langchain** (Node or Python) ou **Haystack** (Python) — ajuda com chunking, retriever, pipelines.
- Tokenizer: **tiktoken** (para chunking por tokens).
- Fuzzy matching / normalization: **fuse.js** or **rapidfuzz** (Python).
- Clustering: **hdbscan**, **scikit-learn**.
- DB client: **@supabase/supabase-js** (Next.js), Postgres + pgvector extension.