Knowledge Hub — Detailed Roadmap

Purpose: A step-by-step, developer-friendly roadmap to build the *AI-Powered Knowledge Hub* (chatbot over company documents) using **ASP.NET Core Web API** backend and a modern frontend (React). This document breaks the project into clear phases, lists required learning, includes API and data models, deployment and security checklists, and developer commands — everything you can hand to a teammate or follow yourself.

> **Note:** Times are intentionally omitted. Treat the roadmap as ordered milestones; move to the next milestone when the previous deliverables are working and tested.

How to use this roadmap

1. Read each **Phase** in order. Complete the checklist items under each phase.
2. Each phase has **Objectives**, **Deliverables**, and **Step-by-step Tasks**.
3. Implement the APIs and tests as you go. Keep a Git branch per major phase (e.g., `feature/mvp-api`, `feature/rag`).

Phase Summary (high level)

Phase	Objective	Key Deliverable
Phase 0 — Prep	Setup dev environment & repo, pick providers	Working repo scaffold, CI placeholder, env file template
Phase 1 — MVP API + UI	Basic upload, text extraction, one-chat flow using cloud LLM (no vector DB)	Upload API, Chat API that calls OpenAI with document text context, simple React UI
Phase 2 — RAG Integration	Add embeddings, vector DB, retrieval, improved prompts	Vector DB upsert/query, RAG pipeline, improved chat accuracy
Phase 3 — Production Hardening	Auth, multi-user, storage, monitoring, CI/CD	Auth flows, Azure deployment, RBAC, monitoring enabled
Phase 4 — Polishing & Advanced	Streaming, multi-tenant, admin UI, cost controls	Admin dashboard, streaming responses, multi-tenant separation

---

Phase 0 — Preparation & Repo

Objective: Set up your development environment, repo, and basic skeleton for backend & frontend.

Deliverables: Repo scaffold with `backend/` and `frontend/` folders, `README.md`, `.env.example`, basic CI skeleton.

Tasks (step-by-step): 1. Create repo on GitHub (`knowledge-hub`). 2. Local machines: install prerequisites. - .NET SDK (recommend .NET 8 or the latest LTS you prefer) - Node.js + npm/yarn - Docker (for vector DB like Qdrant in local dev) - Git 3. Create folders: - `/backend` — ASP.NET Core Web API project - `/frontend` — React + TypeScript project (or Blazor if preferred) 4. Initialize backend:

```
cd backend
dotnet new webapi -n KnowledgeHub.Api
dotnet new sln -n KnowledgeHub
dotnet sln add KnowledgeHub.Api/KnowledgeHub.Api.csproj
git add .
git commit -m "chore: scaffold backend"
```

5. Initialize frontend (React + TypeScript):

```
cd frontend
npm init vite@latest knowledge-hub-frontend -- --template react-ts
cd knowledge-hub-frontend
npm install
```

6. Add `.env.example` with variables (see Environment Variables section later). 7. Create a basic GitHub Actions workflow file placeholder that runs `dotnet build` and `npm install` .

Phase 1 — MVP: Upload + Basic Q&A (no vector DB)

Objective: Implement a minimal working end-to-end flow: upload documents, extract text, and answer user queries by sending relevant document text as context to an LLM (OpenAI API).

Deliverables: - Document upload API (stores original file & extracted text + metadata) - Chat endpoint: concatenate a small set of relevant document text and call OpenAI ChatCompletion - Simple React UI with file upload and chat input

Tasks (detailed):

Backend Tasks

1. Create models & DB (EF Core) for metadata only (no embeddings yet).
2. Tables: `Users`, `Documents`, `Chats`, `Messages`
3. Add file storage: local filesystem in dev, Azure Blob Storage for production.
4. Implement **Document Upload** endpoint:
5. `POST /api/documents` — accepts `multipart/form-data` file, authorId
6. Flow inside:
    - Save file to storage
    - Extract text (PDF/Word). Use libraries:
    - PDF: `iText7` or `PdfPig` or Azure Form Recognizer (if you want higher quality)
    - DOCX: `DocumentFormat.OpenXml` (Open XML SDK)
    - Split extracted text into simple sections (e.g., by headings or every ~1000 characters)
    - Save `Document` metadata and `DocumentSections` to DB
7. Implement **Chat endpoint** (MVP behavior):
8. `POST /api/chat` — payload: `{ userId, question, documentIds?: [] }`
9. Server-side logic:
    - Fetch the chosen documents or user's documents
    - Select the top N sections by basic heuristics (title match, keyword match). *No vector search yet.*
    - Construct a prompt with: System message + context sections + user question
    - Call OpenAI Chat API (ChatCompletions). Return answer.

API examples - Upload - Request `POST /api/documents` (multipart) - Response: `{ id, fileName, status: 'Indexed' }`

- Chat
- Request: `POST /api/chat`

```
{ "userId": "user-123", "question": "What is the leave policy?",
"documentIds": ["doc-1"] }
```

- Response:

```
{ "answer": "..." }
```

Frontend Tasks (MVP UI)

1. Build pages/components:
2. Login (stub or JWT)
3. Dashboard — button to upload document and start chat
4. Upload page — drag & drop file area, show uploaded list
5. Chat page — simple chat UI with input and message bubbles
6. Connect to the backend endpoints using `fetch` or `axios`.
7. Handle loading states & toasts.

Acceptance Criteria (MVP): - You can upload a PDF and extract text. - You can ask a question and receive an answer based on uploaded docs.

Phase 2 — RAG: Embeddings + Vector DB + Retrieval

Objective: Replace naive text selection with a real RAG pipeline: generate embeddings for document chunks, store them in a vector DB, and retrieve relevant chunks by semantic similarity.

Deliverables: - Chunking & embedding pipeline - Vector DB integration (Qdrant/Pinecone/Weaviate/ pgvector) - Chat endpoint that performs embed->query->LLM

Tasks:

Chunking Strategy

- Decide chunk size: e.g., **chunk length 500 tokens (~3000 chars)** with **overlap 50–100 tokens**.
- Store chunk metadata: `documentId`, `chunkIndex`, `text`, `charStart`, `charEnd`, `source_page`.

Embeddings

- Use OpenAI Embeddings API (model: `text-embedding-3-small` or similar). Store embedding vectors in vector DB.
- For local dev, run Qdrant in Docker:

```
docker run -p 6333:6333 qdrant/qdrant
```

- Upsert embeddings to vector DB with chunk metadata.

Vector DB Integration

- Operations required:
- `Upsert(vector, id, metadata)` when indexing
- `Query(vector, topK)` when searching
- `Delete(id)` when removing doc
- If using Pinecone, call its REST API. For Qdrant, use its HTTP API or a .NET client library.

Chat endpoint (RAG)

- Flow:
- Convert user question to embedding.
- Query vector DB for top-K chunks (e.g., top 3–5).
- Build system prompt + include top chunks as `context`.

• Call OpenAI ChatCompletion (or your chosen LLM) with the prompt.
• Return answer and references (which doc/chunk ids were used).

Prompt Template (example)

```
System: You are an assistant that answers only from the provided documents. If
the information is not present, say you don't know.
Context:
---
[Chunk 1 text]
---
[Chunk 2 text]
---
User: {user_question}
```

Important: Limit total context tokens passed to the LLM — trim chunks to keep within token budget.

Frontend Changes - When user asks a question, allow them to optionally choose documents or `All Documents`. - Show which documents the answer was drawn from (small footer: “Sources: HR_Policy.pdf (page 5)”).

Acceptance Criteria: - Semantic search returns relevant chunks and improves answer accuracy over the MVP. - The chat displays sources and chunk references.

Phase 3 — Production Hardening (Auth, Storage, CI/CD)

Objective: Add authentication, multi-user handling, storage in cloud, logging, monitoring, and continuous deployment.

Deliverables: - JWT or Azure AD authentication flows - Blob storage for files - CI/CD pipeline that builds & deploys backend + frontend - Logging & monitoring enabled

Tasks:

Authentication & User Management

• Implement ASP.NET Identity with JWT tokens OR connect to Azure AD for enterprise SSO.
• Enforce role-based access control: roles `Admin`, `User`.
• Implement RBAC checks in endpoints (document read/write, admin document deletion).

Storage

- Replace local file storage with Azure Blob Storage (or S3). Store original files and static assets.

CI/CD

- Create GitHub Actions workflows:
- `backend.yml`: build, test, docker build image, push to container registry.
- `frontend.yml`: build and deploy to Vercel or Azure Static Web Apps.
- `deploy.yml`: deploy backend to Azure App Service or to container service.

Monitoring & Logging

- Integrate Application Insights (Azure) or OpenTelemetry.
- Log: uploads, queries, token usage, errors.
- Create dashboards for user queries and system health.

Secrets Management

- Use Azure Key Vault (or GitHub Secrets) for `OPENAI_API_KEY`, `PINECONE_API_KEY`, DB connection strings.

Acceptance Criteria: - Secure authentication works. - App is deployable via CI. - Monitoring metrics and logs available.

Phase 4 — Polishing & Advanced Features

Objective: Add advanced UX, streaming LLM responses, admin features, and cost-control measures.

Deliverables: - Streaming chat responses (for perceived speed) - Admin dashboard (usage, top queries, re-index controls) - Cost tracking & rate-limits

Tasks: 1. Streaming: use the OpenAI streaming API or server-sent events to stream partial responses to the frontend. 2. Admin dashboard: show token usage per user, top queries, failed queries, ability to delete documents & re-index. 3. Rate limiting: throttling per user to control costs (use in-memory or Redis-based rate limiting). 4. Analytics: store query logs and build dashboards (Grafana or Power BI).

Phase 5 — Enterprise & Future Proofing (Optional)

Options: - Multi-tenant architecture: separate vector namespaces per tenant, tenant-aware auth. - On-premises or VPC deployment for data-sensitive customers. - Hybrid mode: local LLM (Ollama) fallback to avoid sending sensitive snippets to cloud. - Compliance: implement data retention policies and automatic PII redaction.

Data Model (Suggested)

Tables

- `Users` (`Id` , `Name` , `Email` , `PasswordHash` , `Role` , `CreatedAt`)
- `Documents` (`Id` , `UserId` , `FileName` , `BlobUrl` , `Status` , `UploadedAt` , `Size`)
- `DocumentChunks` (`Id` , `DocumentId` , `ChunkIndex` , `Text` , `CharStart` , `CharEnd` , `PageNumber`)
- `Chats` (`Id` , `UserId` , `Title` , `CreatedAt`)
- `Messages` (`Id` , `ChatId` , `Sender` (User|AI), `Text` , `MetaJson` , `CreatedAt`)
- `QueriesLog` (`Id` , `UserId` , `Question` , `UsedChunkIds` , `ResponseTokens` , `CreatedAt`)

Note: Embedding vectors are stored in the vector DB; keep only metadata in your relational DB.

API Design (Suggested Endpoints)

Auth & User - `POST /api/auth/register` — register (name, email, password) - `POST /api/auth/login` — returns JWT - `GET /api/users/me` — profile

Documents - `POST /api/documents` — upload file (multipart), returns document metadata - `GET /api/documents` — list for user - `GET /api/documents/{id}` — metadata - `DELETE /api/documents/{id}` — delete & remove from vector DB - `POST /api/documents/{id}/reindex` — re-chunk and re-index

Chat / RAG - `POST /api/chats` — create a chat - `POST /api/chats/{chatId}/message` — send question; server returns answer - `GET /api/chats/{chatId}/messages` — list chat history

Admin - `GET /api/admin/usage` — token usage, top queries - `POST /api/admin/reindex-all` — reindex entire corpus

---

Prompting & Safety Guidelines

Prompt structure - Use a strict system prompt asking the model to answer only from provided context. Example skeleton:

```
System: You are a helpful assistant. Answer the user's question only using the
supplied context. If the answer is not present, reply: "I don't know — the
documents did not contain that information." Avoid inventing facts.

Context:
[CHUNK 1]
```

```
[CHUNK 2]
...

User: {question}
```

Safety: - If a retrieved chunk contains PII, redact or flag it. - Do not include entire documents as context — only short chunks. - Keep an allowlist/denylist for sensitive words if required.

Environment Variables (example `.env.example`)

```
DOTNET_ENVIRONMENT=Development
ASPNETCORE_URLS=https://localhost:5001
DATABASE_URL=<your-db-connection-string>
BLOB_STORAGE_CONNECTION=<azure-blob-conn>
OPENAI_API_KEY=sk-...
VECTOR_DB_TYPE=qdrant|pinecone|pgvector
VECTOR_DB_URL=http://localhost:6333
VECTOR_DB_API_KEY=
JWT_SECRET=super-secret-key
```

Dev Commands & Tools

- Start backend locally:

```
cd backend
dotnet watch run
```

- Run frontend dev server:

```
cd frontend/knowledge-hub-frontend
npm run dev
```

- Local Qdrant:

```
docker run -p 6333:6333 qdrant/qdrant
```

- EF Core migrations example:

```
dotnet ef migrations add Init
dotnet ef database update
```

CI/CD (GitHub Actions) — small snippet (backend build)

```yaml
name: Backend CI
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Setup .NET
        uses: actions/setup-dotnet@v4
        with:
          dotnet-version: '8.0.x'
      - name: Restore & Build
        run: |
          dotnet restore
          dotnet build --no-restore --configuration Release
      - name: Run tests
        run: dotnet test --no-build --verbosity normal
```

Testing Strategy

- Unit tests: services (document parsing, embedding calls mocked), chunking logic.
- Integration tests: hitting APIs with test files (use a test vector DB namespace).
- E2E tests: run frontend + backend together with known documents and expected answers.
- Mock OpenAI in tests using recorded responses or local mock server.

Monitoring, Logging & Cost Controls

- Log request counts, average response times, OpenAI token consumption per query.
- Add dashboards for: daily queries, top documents, failed requests.
- Implement budget alerts: stop non-admin queries if monthly spend crosses threshold.

Security Checklist

- Transport: enforce HTTPS everywhere.
- Secrets: use Key Vault or secrets manager.
- Rate-limit per-user and global throttles to avoid run-away costs.
- Access control: ensure document read rights checked on every query.
- Data retention: provide admin option to delete user data and purge embeddings for GDPR compliance.

Appendix — Example LLM call (pseudo C#)

```
// Pseudo-code to call OpenAI ChatCompletion after retrieving context
var prompt = $"System: You are an assistant...\nContext:\n{ctx}\nUser:
{question}";
var response = await OpenAiClient.ChatCompletions.CreateAsync(new
ChatCompletionCreateRequest {
    Model = "gpt-4o-mini",
    Messages = new List<Message> {
        new Message("system", "You are an assistant..."),
        new Message("user", prompt)
    }
});
var answer = response.Choices.First().Message.Content;
```

Final checklist (ready-to-check)

- [ ] Repo scaffolded
- [ ] Backend basic API (upload & chat) working
- [ ] Frontend MVP (upload + chat) working
- [ ] Embeddings + vector DB integrated
- [ ] Auth & RBAC implemented
- [ ] CI/CD configured
- [ ] Monitoring & cost controls enabled
- [ ] Admin dashboard & re-index features

If you want this as a downloadable **PDF** or as a **GitHub-ready checklist (issues + labels)**, tell me which and I will export/convert it for you.