MindLink — AI Personalized Learning Companion

> Complete, detailed project plan: architecture, roadmap, DB schema, APIs, frontend & backend scaffolds, AI/ML design (RAG, embeddings), deployment, CI/CD, testing, security, and first-step commands.

---

1. Project overview

Goal: Build an AI-driven personalized learning companion that adapts to each user's learning style, creates custom study paths, generates quizzes, ingests user files (PDFs/notes), and provides analytics and revision schedules. The product emphasizes a personalized experience: the system learns *how* a user learns and adapts content delivery accordingly.

Primary users: Students, lifelong learners, professionals preparing certifications, tutors.

Core value propositions: - Personalized, adaptive learning plans. - Instant explanations and visualizations tailored to learning style. - Auto-generated quizzes and spaced-repetition schedules. - Personal knowledge base (uploaded notes/PDFs) with retrieval-augmented generation (RAG). - Analytics that show strengths/weaknesses and suggested next steps.

---

2. High-level architecture

```
[Frontend (Next.js + Tailwind)]
         |
         v
[API Gateway / Backend (FastAPI or Spring Boot)]
         |
         +--> Auth Service (JWT + OAuth)
         |
         +--> User/Profile Service (Postgres)
         |
         +--> Content Service (Postgres + S3 for files)
         |
         +--> AI Service (Embeddings + RAG + prompt orchestration)
         |
         +--> Recommendation Engine (event-driven, Redis/Vector DB + ML models)
         |
         +--> Notification Service (emails, push)
         |
         +--> Background Worker (Celery/RabbitMQ or Spring + Kafka)
```

```
Supporting systems: PostgreSQL, Redis, Vector DB (Chroma/FAISS), Object Storage
(S3/MinIO), Monitoring (Prometheus/Grafana), CI/CD (GitHub Actions), Docker/K8s
```

Why FastAPI? Fast iteration for AI endpoints, Python has best libs for embeddings & LangChain. If you strongly prefer Java/Spring Boot, you can keep Spring Boot for core APIs and add a Python microservice for AI.

---

3. Tech stack (recommended)

- Frontend: **Next.js (React) + TypeScript + Tailwind CSS** (SSR benefits for SEO), React Query, Zustand for state.
- Backend API: **FastAPI (Python)** — lightweight, async, excellent for ML integration.
- AI/Embeddings service: Python with **LangChain**, **OpenAI** (or local LLMs), **sentence-transformers**, and **Chroma** or **FAISS** for vector DB.
- Database: **PostgreSQL** (relational data), **Redis** (cache, session, rate-limiting).
- Object storage: **AWS S3** or **MinIO** for dev.
- Message Queue & Background Workers: **RabbitMQ + Celery** (if Python) or **Kafka** for higher scale.
- Auth: **JWT** with refresh tokens; OAuth2 social login (Google/Github).
- CI/CD: **GitHub Actions**.
- Containerization: **Docker**; orchestrate with **k3s** or **kubernetes** for prod.
- Monitoring: **Prometheus + Grafana**, **Sentry** for error tracking.

---

4. Features (MVP and beyond)

MVP features (must-have first)

1. User auth & profiles (roles: user, tutor, admin)
2. Upload notes/PDFs and parse text
3. Chat interface with AI to query personal knowledge base
4. Generate topic-wise quizzes and track scores
5. Learning plan generator (based on topics and user schedule)
6. Progress dashboard and weakness heatmap
7. Basic recommendation engine (next topics)

Post-MVP (phase 2+)

- Adaptive content format switching (video vs text vs visual) based on user preference
- Spaced repetition scheduling (SRS) integrated with quiz results
- Collaborative learning: peers share notes, tutor sessions
- Fine-grained personalization (micro-adjustments to learning pace)
- Mobile app (React Native)
- Offline sync, export study packs

---

5. Data model (simplified)

Tables (PostgreSQL)

users

```sql
CREATE TABLE users (
  id UUID PRIMARY KEY,
  name TEXT,
  email TEXT UNIQUE,
  password_hash TEXT,
  role TEXT DEFAULT 'user', -- user, tutor, admin
  preferences JSONB, -- learning style, preferred content type
  created_at TIMESTAMP DEFAULT now()
);
```

topics

```sql
CREATE TABLE topics (
  id UUID PRIMARY KEY,
  name TEXT,
  parent_id UUID NULL,
  metadata JSONB
);
```

documents (uploaded PDFs/notes processed into chunks)

```sql
CREATE TABLE documents (
  id UUID PRIMARY KEY,
  user_id UUID REFERENCES users(id),
  title TEXT,
  s3_key TEXT,
  status TEXT, -- processing, ready
  created_at TIMESTAMP DEFAULT now()
);
```

document_chunks (chunked text with embedding id)

```sql
CREATE TABLE document_chunks (
  id UUID PRIMARY KEY,
  document_id UUID REFERENCES documents(id),
  chunk_text TEXT,
  vector_id TEXT, -- reference to vector DB
  embedding JSONB,
  token_count INT
);
```

quizzes

```
CREATE TABLE quizzes (
  id UUID PRIMARY KEY,
  user_id UUID REFERENCES users(id),
  topic_id UUID REFERENCES topics(id),
  questions JSONB, -- structure of Q/A
  score INT,
  created_at TIMESTAMP DEFAULT now()
);
```

study_plans

```
CREATE TABLE study_plans (
  id UUID PRIMARY KEY,
  user_id UUID REFERENCES users(id),
  schedule JSONB, -- list of sessions, durations
  goals JSONB,
  created_at TIMESTAMP DEFAULT now()
);
```

user_progress

```
CREATE TABLE user_progress (
  id UUID PRIMARY KEY,
  user_id UUID REFERENCES users(id),
  topic_id UUID REFERENCES topics(id),
  strength_score FLOAT,
  last_practiced TIMESTAMP,
  history JSONB
);
```

events (for analytics, recommendations)

```sql
CREATE TABLE events (
  id UUID PRIMARY KEY,
  user_id UUID,
  event_type TEXT,
  payload JSONB,
  created_at TIMESTAMP DEFAULT now()
);
```

---

6. Vector store & RAG design

Why RAG?: Users upload notes/documents. To answer user queries with context from their personal materials, we use embeddings + vector similarity (RAG) to fetch top-k relevant chunks and then call a language model to generate answers grounded in those chunks.

Component choices: - Vector DB: **Chroma** (easy to embed locally) or **FAISS** for large scale, or managed (Pinecone, Weaviate) for production. - Embeddings: **OpenAI embeddings** or local **sentence-transformers** (all-MiniLM) for cost efficiency. - Orchestration: **LangChain** to chain: embed → search → build prompt → call LLM.

Document ingestion pipeline (worker) 1. Upload PDF → store in S3. 2. Background worker pulls PDF, extracts text (PyMuPDF / pdfplumber). 3. Clean & chunk text into ~500-token chunks with overlap. 4. Compute embeddings for each chunk. 5. Insert embeddings & chunk metadata into vector DB; store chunk metadata in `document_chunks`.

Query flow (chat user asks a question) 1. Client sends user query to AI service. 2. AI service computes embedding for the query. 3. Vector DB returns top-k chunks (e.g., k=6) relevant to the query. 4. Construct a prompt with user profile info + retrieved chunks + system instructions. 5. Call LLM (OpenAI/GPT) with the prompt, retrieve answer. 6. Optionally store `events` for analytics and to feed recommendations.

Prompt design tips: - Use short system prompts explaining: "You are a helpful tutor, answer using only the provided documents, cite chunk ids if referencing them." - Avoid hallucination: include instruction to say "I don’t know" when not in sources.

---

7. Recommendation & personalization engine

Data sources: quiz results, document reads, time spent, events, user preferences.

Approach: - Start with heuristic rules: If a user scores <60% in topic A, recommend prerequisite topics and daily micropractice. - Add collaborative filtering (item-based) using event history over time. - Add content embeddings: vector similarity between user-strong topics and other contents to recommend similar materials. - Long-term: train a supervised model (e.g., LightGBM) predicting topic mastery probability given features. Use outputs to schedule SRS.

SRS integration: - Use a simple SM-2 algorithm (Anki-style) to compute next review time based on quiz performance and difficulty.

---

8. APIs (examples)

Auth

- `POST /api/auth/signup` — body: `{name,email,password}` → returns `{access_token, refresh_token}`
- `POST /api/auth/login` — body: `{email,password}`
- `POST /api/auth/refresh` — body: `{refresh_token}`
- `GET /api/auth/me` — returns user profile

Documents

- `POST /api/documents` — multipart upload file -> returns document id
- `GET /api/documents/{id}` — metadata
- `POST /api/documents/{id}/process` — trigger background processing

AI

- `POST /api/ai/chat` — body: `{user_id, prompt, session_id}` → returns `{response, sources[]}`
- `POST /api/ai/generate-quiz` — body: `{user_id, topic_id, difficulty}` → returns `{quiz_id, questions}`
- `POST /api/ai/summarize` — body: `{document_id, options}`

Study plans & Progress

- `POST /api/study-plans` — create plan
- `GET /api/study-plans/{id}` — get plan
- `GET /api/user-progress/{user_id}`

Events & Analytics

- `POST /api/events` — track client events (read, answer, quiz_result)
- `GET /api/analytics/user/{user_id}` — aggregated metrics

---

9. Frontend pages & components

Main pages - Landing / Marketing - Signup / Login - Dashboard (progress summary, next tasks) - Study Plan builder - Document upload & library - Chat / Tutor interface - Quiz player (question/answer UI) - Topic explorer / Search - Settings (preferences) - Admin (manage users, content)

Key UI components - ChatWindow (message list, composer) with streaming LLM responses - DocumentUploader (drag/drop) + progress - QuizPlayer (question display, timer, immediate feedback) - Heatmap/SkillMap (visualize strengths/weaknesses) - Scheduler (calendar view for study plan)

UX details - Let users pick learning style initially: visual, textual, auditory, mixed — store in preferences - Onboarding flow: 5 min setup quiz to gauge level and style - Provide “explain like I’m 5” / “detailed” toggles for the AI responses

---

10. Background tasks & scaling

Tasks to run asynchronously: - Document processing & embeddings - Thumbnail / media processing - Quiz generation - Reindexing/updating vectors - Email notifications & reminders

Scaling notes: - Separate AI-heavy endpoints into their own service with autoscaling (GPU if using local LLMs) or managed APIs (OpenAI). - Cache embeddings and top-k results for repeated queries. - Use horizontal partitioning for vector DB when scale grows.

---

11. Security & privacy

- Authentication: strong password policy, bcrypt/argon2 for hashing, JWT with short expiry + refresh tokens.
- Data protection: encrypt sensitive fields at rest (e.g., PII) and in transit (TLS).
- File security: S3 signed URLs for uploads/downloads.
- User consent: explicit consent for storing personal educational data.
- GDPR: expose endpoints for data export & deletion.
- Rate limiting: protect AI endpoints to avoid abuse & cost spikes.
- Cost control: guard prompts that trigger large token usage; add quotas.

---

12. Testing strategy

- Unit tests: backend services (pytest), frontend components (Jest + React Testing Library)
- Integration tests: API endpoints (TestClient for FastAPI). Use Testcontainers for Postgres in CI.
- E2E tests: Cypress (login flow, upload -> process -> query)
- Load tests: k6 for simulated quiz and chat loads
- ML tests: unit tests for ingestion pipeline; regression tests to ensure no drift in embeddings pipeline

---

13. Observability & Monitoring

- Logs: structured logs (JSON) shipped to ELK or Grafana Loki
- Metrics: Prometheus metrics for request rates, latency, error rates, background jobs
- Tracing: OpenTelemetry distributed tracing for request flows across services

- Alerts: SLOs/SLA, error rate thresholds, high-cost AI usage alerts

---

14. CI/CD and deployment

Local dev: Docker Compose with services: backend, postgres, redis, minio, rabbitmq, chroma (or local vector DB)

`docker-compose.yml` (simple outline)

```
version: '3.8'
services:
  postgres:
    image: postgres:15
    environment:
      POSTGRES_DB: mindlink
      POSTGRES_USER: mindlink
      POSTGRES_PASSWORD: mindlink
    volumes:
      - pgdata:/var/lib/postgresql/data

  redis:
    image: redis:7

  minio:
    image: minio/minio
    command: server /data
    environment:
      MINIO_ROOT_USER: minio
      MINIO_ROOT_PASSWORD: minio123

  backend:
    build: ./backend
    depends_on: [postgres, redis, minio]
    ports: [8000:8000]

  ai-service:
    build: ./ai_service
    depends_on: [postgres]
    ports: [8100:8100]

volumes:
  pgdata:
```

CI (GitHub Actions): tests, lint, build docker images, push to registry, deploy to k8s cluster (or Vercel for frontend + managed backend on Render/AWS).

Deployment options: - Small scale: Render/Heroku for backend + Vercel for frontend + managed vector DB (Pinecone) - Production: AWS/EKS + RDS (Postgres), S3, ElastiCache (Redis), EKS with autoscaling.

---

15. Project folder structure (recommendation)

```
mindlink/
├─ backend/                # FastAPI app
│  ├─ app/
│  │  ├─ main.py
│  │  ├─ api/
│  │  ├─ services/
│  │  ├─ models/
│  │  ├─ db/
│  │  └─ workers/
│  ├─ Dockerfile
│  └─ requirements.txt
├─ ai_service/             # embeddings, langchain orchestration
│  ├─ main.py
│  ├─ ingest.py
│  ├─ vector_store.py
│  └─ Dockerfile
├─ frontend/               # Next.js app
│  ├─ src/
│  ├─ pages/
│  ├─ components/
│  └─ package.json
├─ infra/
│  ├─ docker-compose.yml
│  ├─ k8s/
│  └─ terraform/
└─ docs/
```

---

16. Sample code snippets (get started)

1) FastAPI auth skeleton (`backend/app/main.py`)

```
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
```

```
app = FastAPI()
app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:3000"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

@app.get("/health")
async def health():
    return {"status": "ok"}
```

2) Document ingestion sketch (ai_service/ingest.py)

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
from sentence_transformers import SentenceTransformer
import chromadb

# 1. extract text from pdf (use PyMuPDF / pdfplumber)
# 2. split into chunks
splitter = RecursiveCharacterTextSplitter(chunk_size=800, chunk_overlap=100)
chunks = splitter.split_text(full_text)

# 3. embed
model = SentenceTransformer('all-MiniLM-L6-v2')
embeddings = model.encode(chunks, show_progress_bar=True)

# 4. upsert into chroma
client = chromadb.Client()
collection = client.create_collection("user_docs")
collection.add(ids=ids, documents=chunks, embeddings=embeddings,
metadatas=metadatas)
```

3) Next.js API call example (frontend/src/lib/api.ts)

```
export async function aiChat(userId: string, prompt: string) {
  const res = await fetch('/api/ai/chat', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ user_id: userId, prompt })
  });
  return res.json();
}
```

17. Onboarding & UX flow (first-time user)

1. Signup / Choose learning goals and preferred learning style
2. Short diagnostic quiz (5–10 questions) to estimate current level
3. Ask user to upload notes (optional) or pick topics to study
4. System ingests docs (background) — in meantime show progress
5. Provide first study plan (7-day micro-schedule) + 1 practice quiz

18. Roadmap & milestones (8 major milestones)

Milestone 0 — Setup & scaffolding - Initialize repos (frontend/backend/ai_service) - Docker Compose dev environment - Basic CI pipeline (lint + tests)

Milestone 1 — Auth & user profiles - Implement signup/login, JWT, user preferences - User profile & onboarding flow

Milestone 2 — Document upload & ingestion - Upload UI, background processing, store chunks & vectors - Minimal UI to view uploaded docs

Milestone 3 — Chat + RAG - AI chat endpoint using vector retrieval and LLM - Frontend chat UI with streaming responses

Milestone 4 — Generate quizzes & basic SRS - Quiz generation API - Quiz player UI, store results, update user_progress - Simple SRS scheduler for reviews

Milestone 5 — Study plans & recommendations - Generate study plans based on diagnostics & goals - Recommendation engine (heuristics + embeddings)

Milestone 6 — Analytics & dashboards - Skill heatmap, progress graphs, session logs - Admin tools for content moderation

Milestone 7 — Testing & hardening - Integration/E2E tests, load testing - Monitoring, logging, rate limiting

Milestone 8 — Deployment & scale - Deploy to cloud, configure autoscaling, backups - Add billing/usage monitoring & paid tiers

19. Minimum Viable Product (MVP) checklist

- [ ] Repo scaffolds (frontend, backend, ai_service)
- [ ] Docker Compose with dev services
- [ ] User auth & onboarding
- [ ] Document upload + ingestion pipeline
- [ ] Vector DB + embeddings
- [ ] AI chat with RAG
- [ ] Quiz generation and basic study plan
- [ ] Progress tracking and dashboard

---

20. First-step practical instructions (do this now)

Option A — Preferred (Fast iteration & AI): Use FastAPI + Next.js

Commands to create local scaffold

```
# Root folder
mkdir mindlink && cd mindlink

# Frontend
npx create-next-app@latest frontend --ts
cd frontend
npm install tailwindcss@latest postcss autoprefixer
npx tailwindcss init -p
cd ..

# Backend
python3 -m venv venv
source venv/bin/activate
mkdir backend && cd backend
python -m pip install fastapi uvicorn sqlalchemy alembic asyncpg pydantic boto3
# create backend/app structure
cd ..

# AI service
mkdir ai_service && cd ai_service
python -m pip install langchain sentence_transformers chromadb pymupdf
```

Run dev stack (docker-compose) - Create `infra/docker-compose.yml` (use the sample from section 14) - `docker compose up --build`

Verify: - Backend health: `curl http://localhost:8000/health` - Frontend: `npm run dev` in `frontend` and open `http://localhost:3000`

21. Seed data & Postman collection outline

Seed ideas: sample user accounts, a few topics (Math: Algebra, Calculus), sample PDFs (short notes), sample quizzes.

Postman collection endpoints (suggested) - Auth: signup, login, me - Documents: upload, process, list - AI: chat, summary, generate-quiz - Study plans: create, list - Progress: get user progress

22. Cost considerations & tips

- OpenAI/commercial LLM costs can add up — cache responses, use smaller models for embeddings, throttle queries.
- Consider a hybrid: open-source embeddings + OpenAI for generation; or use cheaper models for smaller responses.
- Use managed vector DBs (Pinecone) for production to reduce infra complexity if budget allows.

23. Privacy, ethics & moderation

- Provide clear terms: how user data & documents are used.
- Avoid using user-uploaded data for model fine-tuning without explicit opt-in.
- Implement content moderation pipeline for abusive content.

24. Stretch goals (future)

- Peer tutoring marketplace (tutors, booking, live sessions)
- Auto-generate visual aids (diagrams) using AI image models
- Adaptive lesson micro-curricula that change mid-lesson based on user confusion signals (analysis of time per step, wrong answers)
- Institutional dashboards for teachers to push content to classrooms

25. Mentorship / Collaboration model

If you want, I can: - Generate the **full repo scaffolds** (skeleton code for backend, ai_service, frontend) in files you can copy. - Provide **complete implementations** for each milestone (controllers, services, LangChain pipelines, React pages) step-by-step. - Debug errors you paste here and suggest fixes and PR-ready code.

26. Next choices for you (pick one)

- **A — Scaffold Backend + AI service** (FastAPI skeleton, Dockerfile, ingestion scripts) *(recommended)*
- **B — Scaffold Frontend** (Next.js + Tailwind skeleton, pages for auth, upload, chat)
- **C — Full Docker Compose + sample data** (dev env ready to run)
- **D — Deliver Milestone 1 complete** (Auth, onboarding, diagnostic quiz)

Tell me which option (A/B/C/D) and I will **generate the exact files and commands** for that step immediately.

End of document.