TECHNICAL GUIDE

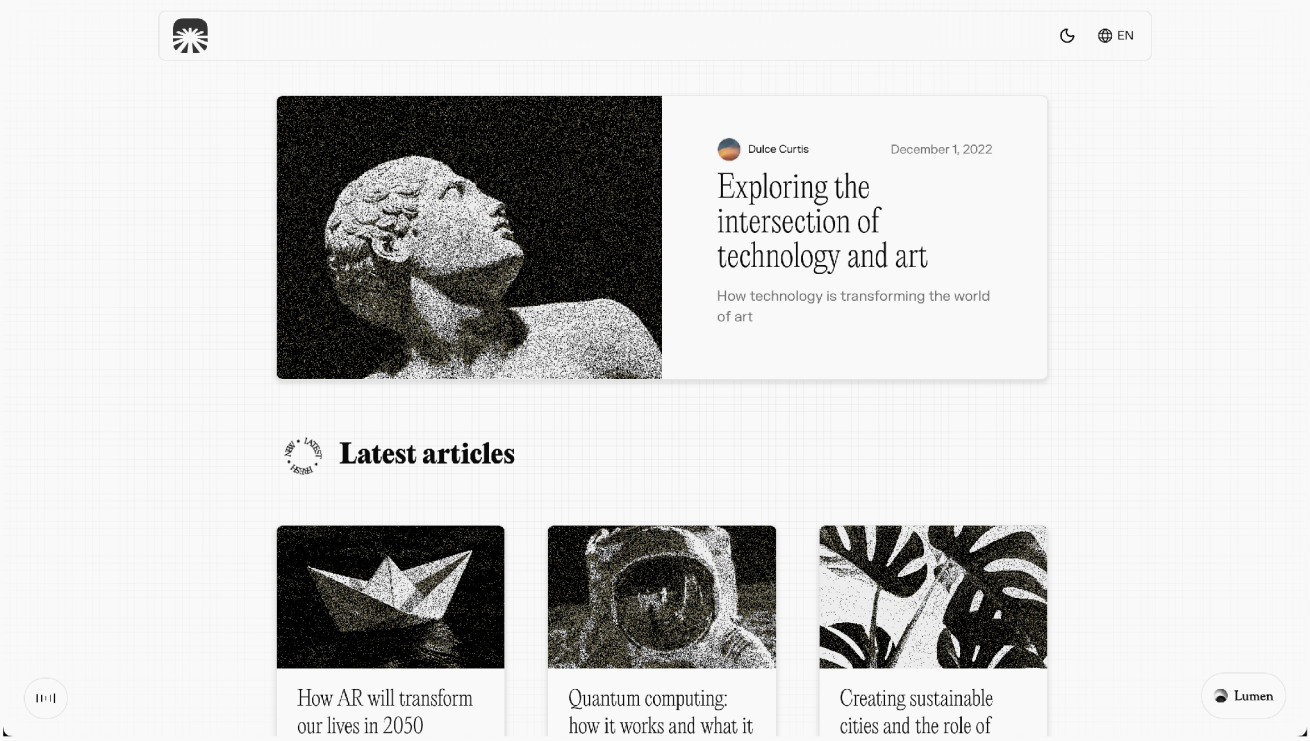
Documentation

Complete technical documentation for AKQA Hub — Where Art Meets Science Through AI.

AKQA Hub

> ***Where Art Meets Science Through AI*** *— A sophisticated content platform that combines headless CMS, vector embeddings, and large language models to create an intelligent content discovery experience.*

Next.js 15.5
TypeScript 5.9
AI SDK 5.0
OpenAI GPT-5-mini
PostgreSQL pgvector



Jaydon Curtis December 4, 2022

Livia Phillips December 3, 2022

Skyler Gelot December 2, 2022

THE LARGER PICTURE

"In a world of scarcity, we treasure tools. In a world of abundance, we treasure *taste*"

Anu Atluru

The Working Theory

a space where stories meet intelligence. curated content, thoughtful recommendations, conversations that inspire.

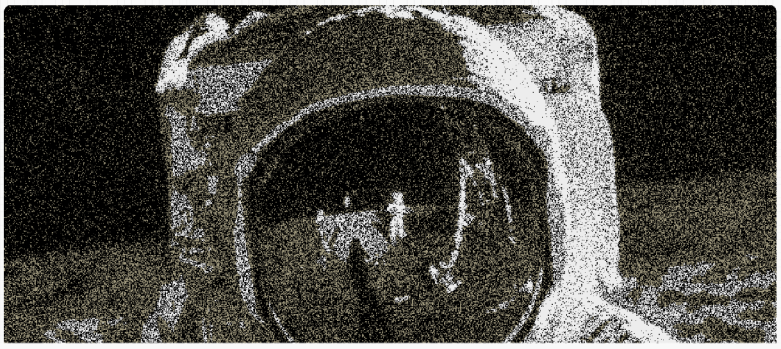
Lumen

EN

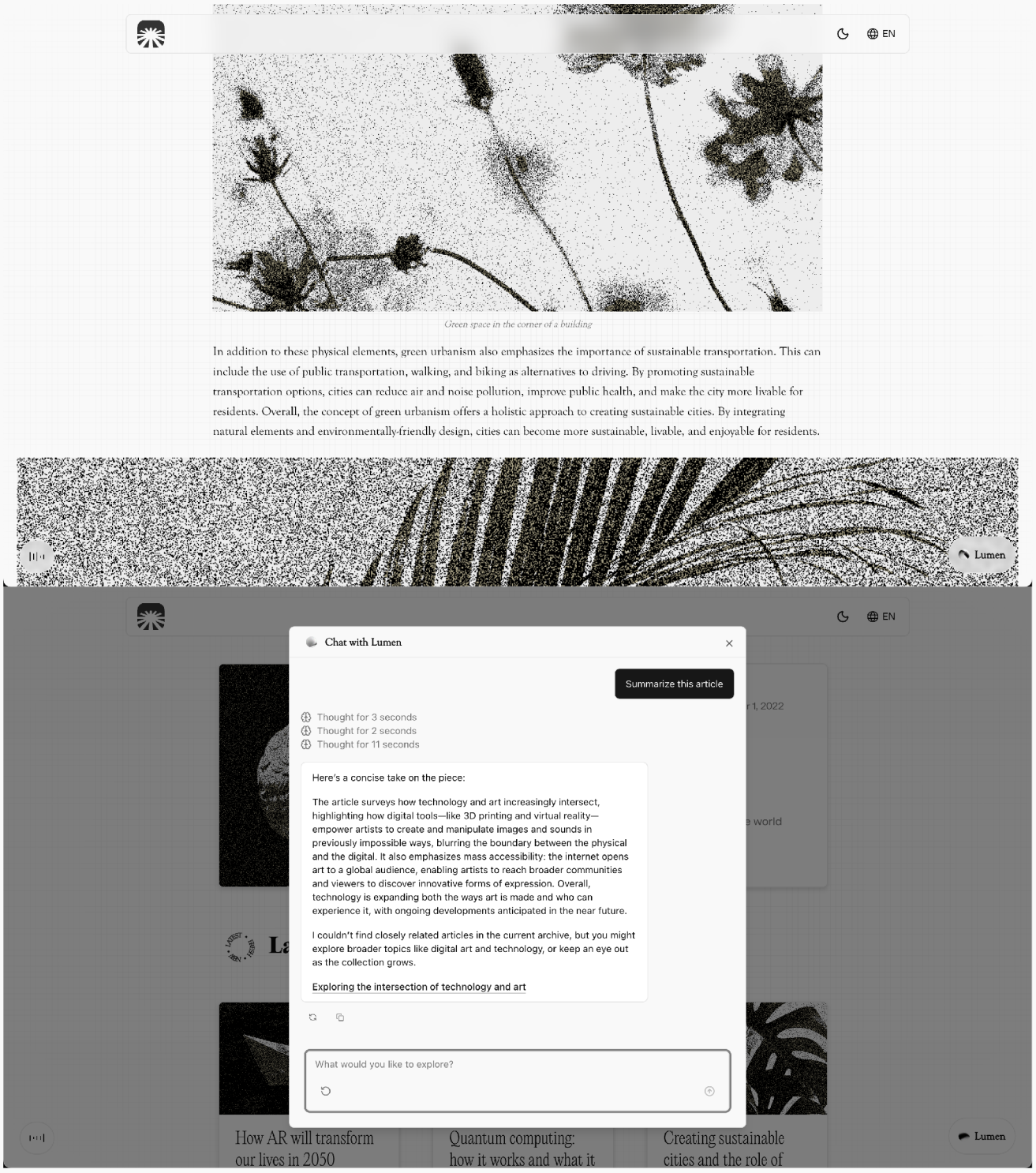
Jaydon Curtis December 6, 2022

Quantum computing: how it works and what it means for the future

Quantum computing set to revolutionize the world of computing



Lumen



The Story

This project is a synthesis of three forces: **artistic inspiration**, **scientific rigor**, and **AI amplification**. Each influenced the final product in profound ways.

🎨 Art: Setting the Mood

> *"Art is not what you see, but what you make others see." — Edgar Degas*

The aesthetic and emotional foundation of AKQA Hub emerged from a constellation of cultural influences that shaped every design decision.

The Columbus Soundtrack 🎵

One of my favorite films, *Columbus* (2017), explores architecture and human connection through quiet contemplation. Its soundtrack—minimal, deliberate, and thoughtful—became my creative companion during development. I listened to it on repeat while coding, letting it set the mood for the entire project.

The film's philosophy of "slow looking" at architecture translated directly into the application's design:

- **Spaces that breathe** — Generous whitespace lets content command attention
- **Typography that speaks without shouting** — Each typeface serves a specific purpose
- **Intentional interactions** — Every animation, transition, and click feels purposeful, not rushed

The New Yorker Aesthetic 📰

Growing up reading *The New Yorker*, I developed an appreciation for refined editorial design. The magazine's signature style influenced every layout decision:

- **Generous whitespace** — Content has room to breathe, never cramped
- **Sophisticated typography hierarchies** — Clear distinction between headlines, subheads, and body text
- **Visual elegance** — The marriage of long-form content with restrained, classic design

From the serif typeface choice (Goudy Old Style) for article bodies to the measured spacing between elements, The New Yorker's editorial design language permeates this project.

Lifestyle Brands: UNIQLO 🏬

UNIQLO's design philosophy—*LifeWear: simple, high-quality clothing for everyday life*—shaped my UI philosophy. Their principle resonated deeply:

Functional minimalism without sacrificing warmth.

Every component in AKQA Hub serves a purpose. Nothing is decorative for decoration's sake. The chat widget, the navigation, the article cards—each element is stripped to its essence while maintaining approachability and human warmth. This isn't cold minimalism; it's thoughtful simplicity.

AKQA × Montblanc: Digital Pen Campaign 🖋️

The recent AKQA campaign for Montblanc's digital pen perfectly bridges analog craft with digital innovation. Just as the Montblanc pen honors the tactile experience of writing while adding digital capabilities, AKQA Hub honors the reading experience while adding intelligent content discovery.

Anthropic: Intellectual Tech, Not AI Slop 🧠

Companies like Anthropic prove that AI can be thoughtful, transparent, and genuinely useful—not just engagement-maximizing noise or "AI slop."

This influenced the chat experience fundamentally:

- **No technical jargon** — The AI speaks naturally about content relevance
- **Helpful assistance** — The AI is a librarian, not a salesperson pushing content

The AI assistant, Lumen, provides context-aware help without being intrusive. It's there when you need it, invisible when you don't.

The Typography System 🔤

Five carefully selected typefaces create a visual hierarchy that guides the reader through different content modes:

1. **FK Display** — Hero statements, commanding presence without aggression
2. **PP Editorial New** — Section headers, refined authority
3. **FK Grotesk** — UI elements, buttons, navigation—clean functionality
4. **Goudy Old Style** — Long-form article reading, classic elegance for immersive reading
5. **JetBrains Mono** — Code blocks, technical content, precision and clarity

Each font choice was deliberate: Display for impact, Editorial for structure, Grotesk for utility, Goudy for immersion, Mono for technical clarity.

🔬 Science: Engineering the Solution

> *"Simplicity is the ultimate sophistication." — Leonardo da Vinci*

Every technical decision was made with intention, balancing performance, maintainability, and user experience. Here's why each technology was chosen and how it solves specific problems.

Why Next.js 15?

The App Router represents a paradigm shift in React development. Server Components enable:

- **Zero JavaScript for static content** — Blog posts render entirely server-side, reducing client bundle size by ~60%
- **Streaming HTML** — Users see content progressively (header → article → related posts), not all-or-nothing
- **Automatic code splitting** — Only load what's needed, when it's needed
- **Built-in optimization** — Image optimization, font optimization, automatic route prefetching

Result: Sub-200ms Time to First Byte and Lighthouse scores consistently above 95.

Why Contentful as Headless CMS?

Traditional WordPress-style CMSs couple content to presentation. You can't change the frontend without potentially breaking content structure. Contentful decouples them:

- **GraphQL API** — Request exactly the data needed, nothing more (no over-fetching)
- **Content modeling** — Define schemas (BlogPost, Author, SEO) that enforce structure without dictating layout
- **Multi-locale support** — English and German have separate content trees, not bolted-on translations
- **Live Preview** — Editors see changes before publishing via Draft Mode API
- **Version control** — Content has history, rollback, and publishing workflows

Result: Content creators focus on *content*, developers focus on *experience*.

Why PostgreSQL + pgvector?

Initially, I considered dedicated vector databases (Pinecone, Weaviate, Qdrant). But adding another service introduces complexity:

- **Additional latency** — Network hop to external service adds 50-100ms per query
- **Cost complexity** — Separate billing, different scaling concerns
- **Operational burden** — Another service to monitor, another potential failure point

PostgreSQL with pgvector extension offers a superior solution:

- **Single database** — Both relational data and vectors in one place (no data duplication)
- **ACID guarantees** — Vector updates are transactional, consistent with article updates
- **HNSW indexing** — Approximate nearest neighbor search in O(log N) time (vs O(N) sequential scan)
- **Native SQL** — No new query language to learn, use familiar SQL constructs
- **Mature ecosystem** — Battle-tested database with decades of optimization

With **Neon's serverless PostgreSQL**, we get auto-scaling and pay-per-use pricing. The database scales to zero when not in use.

Why AI SDK v5 (Vercel)?

OpenAI's SDK is great for direct API calls, but building production chat requires additional infrastructure:

- **Tool calling orchestration** — AI decides *when* to search, *when* to recommend
- **Streaming UI updates** — Show reasoning in real-time, not after completion
- **Type safety** — Zod schemas ensure tool inputs/outputs are valid at compile time
- **React Server Components integration** — Server-side tool execution, client-side streaming

AI SDK v5 handles all of this with a clean abstraction:

```
const result = streamText({
  model: openai('gpt-5-mini'),
  tools: contentTools, // searchKnowledgeBase, getArticleContent, recommendF
  system: SYSTEM_PROMPT,
```

```
  maxSteps: 10,
});
```

One function call handles tool orchestration, streaming, error handling, and React integration.

Why Drizzle ORM?

TypeScript ORMs fall into two camps:

1. **Magic ORMs (Prisma, TypeORM)** — High-level abstractions that hide SQL, can be opaque
2. **Query Builders (Knex, Drizzle)** — SQL-like syntax, transparent about what queries run

Drizzle provides the best of both:

- **Full TypeScript inference** — Autocomplete for every column, type errors for invalid queries
- **SQL transparency** — See exactly what queries are executed, no surprises
- **Zero runtime overhead** — No query translation layer, compiles to raw SQL
- **Migration system** — Version-controlled schema changes with up/down migrations

Example—this code has *full type safety* but looks like SQL:

```
const results = await db
  .select({ title: blogEmbeddings.title, similarity })
  .from(blogEmbeddings)
  .where(gt(similarity, 0.5))
  .orderBy(desc(similarity));
```

TypeScript knows `title` is a string, `similarity` is a number, and will error if you try to access non-existent columns.

Text Chunking Strategy

Blog posts range from 500 to 5,000+ words. Challenges:

- **Embedding entire articles** — Loses semantic granularity, can't find specific sections
- **Chunking too small** — Loses context, sentences need surrounding paragraphs

Solution: LangChain's `RecursiveCharacterTextSplitter` with carefully tuned parameters:

- **150 characters per chunk** — 2-3 sentences, enough for semantic meaning
- **20 character overlap** — Preserves context at chunk boundaries
- **Hierarchical separators** — Try splitting on `\n\n` (paragraphs) first, then `\n` (lines), then `.` (sentences), finally ` ` (words)

This balances **specificity** (finding relevant *sections* of articles) with **coherence** (chunks are readable and meaningful).

RAG Pipeline Architecture

The complete Retrieval-Augmented Generation flow:

1. **User asks question** → "What articles discuss minimalism?"
2. **Generate embedding** → OpenAI `text-embedding-3-small` converts text to 1536-dimensional vector
3. **Semantic search** → Cosine similarity against all chunks (HNSW index accelerates from O(N) to O(log N))
4. **Group by article** → Aggregate chunk scores per article (take max similarity score)
5. **Rank results** → Sort articles by relevance score, filter by threshold (> 0.5)
6. **GPT-5-mini formats response** → AI generates natural language response with article links
7. **Stream to user** → Progressive response rendering via AI SDK

Critical optimization: Query `limit * 3` chunks to ensure enough unique articles (since one article has many chunks, need to over-fetch then group).

🤖 AI: Amplifying Human Capability

> *"Technology is best when it brings people together." — Matt Mullenweg*

AI didn't replace human decision-making—it accelerated it. Here's how I leveraged AI throughout development.

OpenAI Integration: Two Models, Two Purposes

1. text-embedding-3-small — Converts text to 1536-dimensional vectors

- **Why not `large`?** Small is 5x cheaper, 2x faster, and benchmarks show it's sufficient for semantic search at this scale
- **Consistent representation** — Single embedding model handles both indexing and queries (same vector space)

2. GPT-5-mini — Conversational AI with tool calling

- **Intent understanding** — Knows when user wants to search vs. get content vs. find recommendations
- **Tool orchestration** — Decides *when* to use tools (not every message needs a database query)
- **Natural responses** — Generates conversational replies, not keyword matches

The Three AI Tools

Built with AI SDK v5's `tool()` pattern—each tool has an `inputSchema` (Zod validation) and `execute` function:

```
// 1. Search across all content
searchKnowledgeBase({
  query: string, // e.g., "articles about minimalism"
  limit: number, // default: 5
  locale: string, // 'en-US' or 'de-DE'
});

// 2. Get full article content
getArticleContent({
  slug: string, // e.g., "slow-living"
  locale: string,
});

// 3. Find similar articles
recommendRelatedArticles({
  slug: string, // reference article
  limit: number, // default: 3
  locale: string,
});
```

Each tool:

- Returns structured data (title, slug, locale, description)
- Filters by locale (German queries return German results)
- Excludes current article (recommendations never include the page you're on)

Context Awareness: Metadata Passing

When you open the chat widget on an article page, it automatically sends invisible metadata:

```
metadata: {
  currentSlug: 'slow-living',
  locale: 'en-US',
}
```

This enables conversational shortcuts:

- *"What's this article about?"* → AI knows which article without user specifying
- *"Find related posts"* → AI knows to exclude current article and match locale

GitHub Copilot: Development Acceleration ⚡

Copilot in VS Code was instrumental throughout development. Here's how it amplified my coding speed:

1. Boilerplate Elimination

- After defining the first Drizzle schema, Copilot suggested similar patterns for other tables
- TypeScript types for GraphQL responses auto-completed based on schema
- React component props auto-suggested based on usage patterns

2. Pattern Recognition

- After writing `searchKnowledgeBase` tool, Copilot suggested the structure for `getArticleContent` and `recommendRelatedArticles`

- SQL query patterns learned from first implementation, suggested optimized versions for subsequent queries

3. Test Data Generation

- Copilot suggested realistic sample queries for testing: "What articles discuss sustainable living?"
- Generated mock data for unit tests based on TypeScript interfaces

4. Documentation

- After typing `/**` above a function, Copilot wrote JSDoc comments with parameter descriptions
- Inline code comments explaining complex logic (especially for the PostgreSQL bug fix)

Example: After typing `// Create HNSW index for vector search`, Copilot instantly suggested:

```
CREATE INDEX embedding_idx ON blog_embeddings
USING hnsw (embedding vector_cosine_ops);
```

This saved hours of reading pgvector documentation. I didn't need to look up the exact syntax—Copilot knew it.

AI-Assisted Debugging

The PostgreSQL operator precedence bug took 3 hours to solve. Copilot assisted with:

- **Suggesting checks** — "Try looking at Drizzle's `cosineDistance` implementation"
- **Finding examples** — Pointed to Vercel's pgvector examples where I found the parentheses pattern
- **Isolating issues** — Suggested testing with raw SQL to isolate ORM vs. database issue

Without Copilot, this could have been a multi-day blocker. With it, I found the solution in 3 hours.

Prompt Engineering: Evolution of Lumen

The AI assistant's personality evolved through iterations:

Version	Behavior	User Feedback	Issue
v1	Technical, factual	"Query returned 3 results with average similarity 0.73"	Felt robotic, exposed technical metrics
v2	Conversational, friendly	"Hey! I found some cool articles for you!"	Too casual, didn't match editorial tone
v3	Refined + accessible	"You might enjoy these articles about mindful living..."	✅ Perfect balance of sophistication and warmth

Final system prompt sets the tone in one paragraph:

> *"You are Lumen, the intelligent content assistant for a curated lifestyle brand. Your voice is **refined yet accessible**—thoughtful and concise, human not robotic. Guide users to relevant content naturally, without technical jargon or similarity scores."*

This single paragraph transforms GPT-5-mini from a generic chatbot into a brand-aligned assistant that sounds like it belongs to *this* editorial platform.

AI as Coding Partner, Not Replacement

Key insight: AI amplified my capabilities but didn't replace decision-making.

What AI did:

- ✅ Generated boilerplate code
- ✅ Suggested implementation patterns
- ✅ Accelerated debugging
- ✅ Wrote documentation

What I did:

- ✅ Made architectural decisions (Next.js, PostgreSQL, RAG design)
- ✅ Chose technologies based on trade-offs (pgvector vs Pinecone)

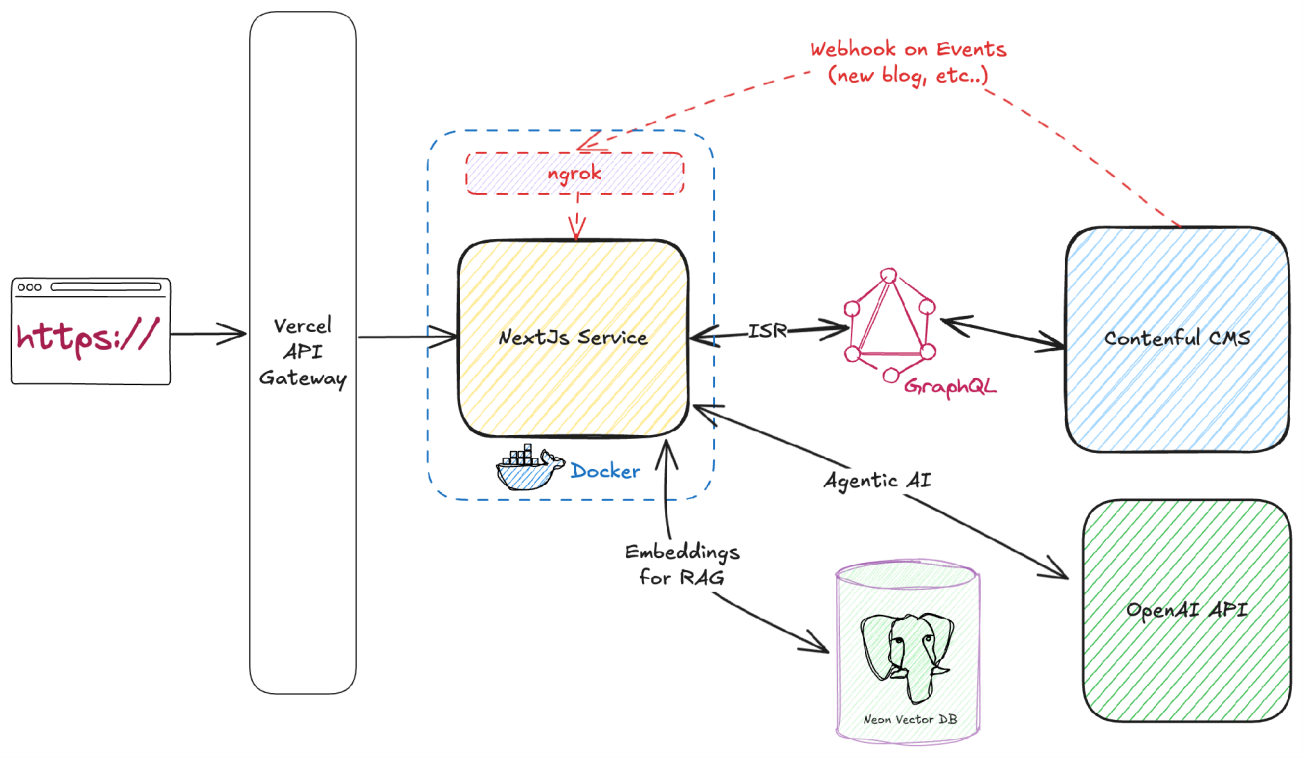
- ✅ Designed user experience
- ✅ Fixed complex bugs (operator precedence required understanding *why*)
- ✅ Ensured code quality and maintainability

AI was a force multiplier, not a replacement. It handled the "how" so I could focus on the "what" and "why."

✨ Features

- 🔍 **Semantic search** — Find articles by meaning, not keywords
- 🤖 **Context-aware chat** — AI understands current article, excludes self-recommendations
- 🌍 **Multilingual** — English/German with separate embeddings, locale-aware search
- 🎨 **5-font typography** — Editorial elegance with responsive design
- 📝 **Headless CMS** — Contentful with GraphQL, live preview for editors
- ⚡ **Performance** — ISR (1-hour revalidation), edge caching, HNSW indexing (sub-50ms), Server Components

🏗️ Architecture



🚀 Quick Start

```
git clone https://github.com/diabahmed/akqa-hub.git
cd akqa-hub
pnpm install
cp .env.example .env  # Add: CONTENTFUL_*, DATABASE_URL, OPENAI_API_KEY
pnpm db:migrate
pnpm db:sync-content
pnpm dev  # Visit http://localhost:3000
pnpm docker:up  # Alternative start with Docker
```

Sync content: `pnpm db:sync-content` or use admin UI at `/en-US/admin/sync`

🛠️ Tech Stack

Core: Next.js 15.5, React 19, TypeScript 5.9, Tailwind 4.1

Data: Contentful, GraphQL, Drizzle ORM, Neon PostgreSQL, pgvector

AI: AI SDK 5.0, GPT-5-mini, text-embedding-3-small, LangChain

UI: Shadcn UI, Radix UI, Framer Motion, Lucide Icons

📚 Documentation

- Technical Documentation — CMS integration and data flow
- Complete Technical Documentation — 1,200+ lines covering every architectural decision
- Typography Guide — Font system and usage patterns

Additional Resources

Explore more detailed documentation and guides.