Brainer: A Multilingual, Voice-Driven AI Note-Taking System with Semantic Recall

Dishank Chauhan
Independent Researcher
Full Stack Developer
India
https://github.com/DishankChauhan

***Abstract*—Digital note-taking tools have long served as simple storage bins for text. This paper introduces Brainer, our approach to building a more intelligent system that actively helps users manage their knowledge. Brainer provides a voice-first interface that accurately transcribes both English and Hindi in real-time. It uses an AI pipeline to automatically summarize notes and add smart tags. Most importantly, it features a semantic recall function, letting users find notes by meaning, not just keywords. We built Brainer on a scalable cloud architecture to show how today's AI services can be combined to create a truly intuitive knowledge tool. This paper covers our architecture, the AI pipeline, the challenges we faced with multilingual support, and a full performance evaluation.**

***Index Terms*—note-taking, artificial intelligence, voice recognition, natural language processing, semantic search, multilingual systems, AWS, GPT-4**

I. Introduction

Most note-taking apps today are little more than digital filing cabinets. You can put things in, but getting them out in a meaningful way is still a manual chore. The burden of organizing, connecting, and remembering information falls entirely on the user. We saw an opportunity to change this, especially with the recent advances in Large Language Models (LLMs) and cloud AI.

Our goal was to build a note-taking assistant that doesn't just store notes, but actually understands them. We focused on three main goals:

1) **Natural Input**: Make capturing ideas effortless through voice. Our contribution here is a real-time, bilingual voice transcription pipeline that works for both English and Hindi with high accuracy.
2) **Automated Processing**: Automate the tedious work of summarizing and tagging. Our AI pipeline handles this by generating concise summaries, pulling out key points, and suggesting relevant tags.
3) **Intelligent Retrieval**: Create a search that works like a human brain, based on concepts, not just words. We accomplished this with a semantic recall feature that finds related notes based on conceptual similarity.

Brainer is the result of that effort. It's our case study in building a modern web app powered by AI, stitching together services from AWS, OpenAI, and Firebase to create something that feels cohesive and performs well.

II. System Architecture

To build Brainer, we chose a layered architecture. This helped us keep the system modular and easier to manage as it grew. You can think of it in four main parts: the frontend the user interacts with, the AI layer where the thinking happens, our own backend services, and the database that holds everything together. The overall design is shown in Figure 1.

A. Frontend

We built the frontend as a single-page application using Next.js 14 with TypeScript. Using the App Router helped us with server-centric data fetching. The UI itself was built with Tailwind CSS and a set of our own custom components. For state management, we used a mix of React hooks and the Context API for global things like user authentication. For more complex server-side data, we wrote custom hooks to handle the fetching, caching, and real-time updates.

B. AI & Voice Layer

This is where the magic of Brainer happens. It's not one single thing, but a collection of services that our backend coordinates to process user input.

- We use the browser's **Web Speech API** to capture the user's voice, and then stream it to our backend for heavier processing.
- **AWS Transcribe** does the heavy lifting for speech-to-text, giving us accurate transcriptions for long audio files.
- **OpenAI's GPT-4o-mini** model is our go-to for summarizing text, generating tags, and other language understanding tasks.
- The **Web Speech Synthesis API** gives the assistant a voice, providing audio feedback to the user.

C. Backend Services

Our backend is a collection of serverless functions built using Next.js API Routes. We went this route because it integrates cleanly with our frontend and we can deploy it easily on platforms like Vercel that handle auto-scaling. The main services are:

- An **API Gateway** that acts as the front door for all requests from the client.

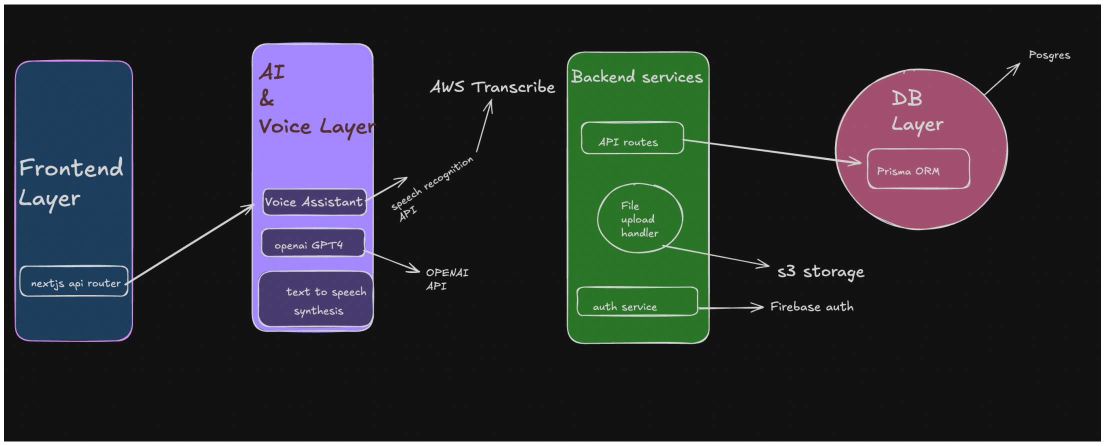

Fig. 1. High-level system architecture of Brainer, illustrating the interaction between the frontend, backend, AI services, and database layers.

- A **File Upload Handler** that securely takes in audio and image files. It creates pre-signed URLs so files go directly to AWS S3, which keeps the load off our backend.
- A **Real-time Polling Service** to give the user status updates for long jobs, like when a long audio file is being transcribed.
- **Authentication Middleware** that checks for a Firebase Authentication token to protect sensitive API routes.

D. *Database Layer*

For our database, we chose PostgreSQL because it's robust, reliable, and has some powerful extensions we needed.

- We use **Prisma ORM** for all our database interactions. This gives us type-safe access and makes managing the schema and migrations much simpler.
- The main **PostgreSQL** database is a managed instance that stores all the structured data: user info, note text, metadata, and tags.
- To handle semantic search, we use the **pgvector** extension right inside PostgreSQL. This lets us store and index the high-dimensional vectors we create from the notes.

E. *External Services*

We didn't build everything from scratch. Brainer stands on the shoulders of several managed cloud services:

- **AWS Transcribe** for speech-to-text.
- **AWS S3** for storing all uploaded media.
- **OpenAI API** for the LLM intelligence.
- **Firebase Authentication** for user logins.
- **Vercel** for hosting, continuous deployment, and running our serverless functions.

III. AI AND NLP PIPELINE

The core of Brainer's intelligence is its AI and NLP pipeline. This is the process that takes raw input, like a voice memo or a picture of a whiteboard, and turns it into a structured, useful note. The process involves several steps.

A. *Real-time Transcription*

When a user uploads an audio file, our backend starts a job with AWS Transcribe. For live sessions, we use 'StartStreamTranscription', but for longer, pre-recorded files, we use 'StartTranscriptionJob' to process them in the background. We have language auto-detection turned on, which is key for our bilingual users. Transcribe sends us back the full text, but also gives us word-level confidence scores and timestamps, which we plan to use to build more UI features later.

B. *Prompt Engineering & Summarization*

After we get the text from Transcribe, we send it to OpenAI's GPT-4o-mini model. We specifically chose GPT-4o-mini because it offered a great balance between the power of a model like GPT-4 and the speed and lower cost of smaller models. Getting the output we wanted required some careful prompt engineering. We designed a prompt that tells the model to return a clean JSON object with everything we need in one go.

Listing 1. Summarization and Tagging Prompt

```
As an expert note analysis assistant, process the
    ↪ following transcript.
Your response must be a single, valid JSON object.
    ↪ Do not include any text outside of the JSON.

The JSON object must have three keys:
1. "title": A concise, descriptive title for the
    ↪ note (max 10 words).
2. "summary": A bulleted list of the 3-5 most
    ↪ important points.
3. "tags": An array of 3-5 relevant lowercase
    ↪ keywords for categorization.

Transcript:
"""
{{transcript_text}}
"""
```

This trick of asking for structured JSON saves us a lot of parsing headaches and makes the output reliable.

C. Semantic Embeddings & Memory Recall

To make semantic search possible, we have to create a ”fingerprint” of every note’s meaning.

1) We take the note’s title and its full text and combine them into one piece of text.
2) We send this text to an embedding model (OpenAI’s ‘text-embedding-ada-002‘) which returns a high-dimensional vector.
3) We then save this vector in a special ‘vector‘ column in our ‘Note‘ table in PostgreSQL, using the ‘pgvector‘ extension.

So, when a user is looking at a note, we grab that note’s vector and run a similarity search to find other notes that are conceptually related. The query looks for the top 5 notes with the closest vectors, calculated using cosine similarity.

Listing 2. Semantic Similarity Query

```
SELECT id, title, 1 - (embedding <=> :query_vector)
    ↪ AS similarity
FROM "Note"
WHERE id != :current_note_id
ORDER BY similarity DESC
LIMIT 5;
```

This is how Brainer can show you relevant information you might have forgotten about.

D. OCR Pipeline

We wanted Brainer to read images too. The pipeline for that is quite similar to the audio one.

1) The user uploads an image to an S3 bucket.
2) Our backend starts an analysis job with a service like AWS Textract to pull out all the text in the image.
3) That extracted text then goes through the exact same summarization and embedding pipeline as our transcribed audio. This makes everything, regardless of its source, equally intelligent.

IV. Multilingual Design

Building a system for both English and Hindi users brought some interesting challenges.

- **Language Detection**: We relied heavily on AWS Transcribe’s ability to automatically identify the language in an audio file. For plain text, we use a small client-side library to guess the language so we can give the AI the right context.
- **Script Handling**: We had to make sure our entire system was happy with Devanagari script. This meant setting our database to use UTF-8 encoding and choosing fonts for the UI that could render the script properly.
- **Localized Responses**: We couldn’t use the same AI prompt for both languages. We dynamically adjust the prompt based on the language of the note to make sure the summaries and tags it generates feel natural and contextually correct.

V. Backend Infrastructure & Deployment

A. API Design

We tried to keep our API design simple and RESTful. For example, a ‘GET‘ request to ‘/api/notes‘ gets all the notes, while a ‘POST‘ to the same URL creates a new one. Next.js’s file-based routing made this very intuitive to set up.

B. Authentication

We use Firebase Authentication for user logins. The process is pretty standard:

1) The user signs in on the frontend using the Firebase SDK.
2) Firebase gives back a JWT (ID token).
3) We attach that token to every API request that needs authentication.
4) A middleware on our backend uses the Firebase Admin SDK to check if the token is valid before letting the request go through.

C. Prisma + PostgreSQL Schema

Prisma was a huge help in managing our database schema. Here’s a simplified look at our core data models.

Listing 3. Simplified Prisma Schema for Notes

```
model User {
  id      String @id @default(cuid())
  email   String @unique
  notes   Note[]
}

model Note {
  id            String    @id @default(cuid())
  title         String
  content       String?
  summary       String?
  createdAt     DateTime  @default(now())
  updatedAt     DateTime  @updatedAt
  owner         User      @relation(fields: [ownerId
      ↪ ], references: [id])
  ownerId       String
  tags          Tag[]
  embedding     Unsupported("vector(1536)")? // for
      ↪ pgvector
}

model Tag {
  id       String @id @default(cuid())
  name     String @unique
  notes    Note[]
}
```

That ‘Unsupported(”vector(1536)”)‘ part is the special syntax Prisma uses to work with the ‘pgvector‘ extension.

D. AWS S3 for Media Storage

To handle large file uploads without bogging down our servers, we have the client upload them directly to AWS S3 using pre-signed URLs that our backend generates. This is much more efficient and scalable. We also have S3 bucket policies in place to make sure users can only get to their own files.

E. Real-time Polling

Transcription can sometimes take a few minutes, so we needed a way to tell the user what was happening. We considered WebSockets, but decided on a simpler long-polling approach.

- **The Trade-off**: WebSockets are faster but add a lot of complexity, especially in a serverless setup. Polling is stateless and more resilient, and for this use case, the slightly higher latency was perfectly fine.
- **How it Works**: The client just calls an API endpoint (‘/api/transcribe/status/jobId‘) every few seconds. The backend checks the status with AWS and sends back the result. It’s simple, cheap, and gets the job done.

F. Scalability

We designed the system with scalability in mind from day one. By using serverless functions on Vercel and managed services like S3 and Transcribe, the system can handle traffic spikes without any manual intervention. Our database is also a managed instance that we can scale up with a few clicks if we need to.

VI. Evaluation

A system with these kinds of claims needs solid proof that it works. We ran a series of tests to measure Brainer’s performance in the real world. We looked at transcription accuracy, how fast it could generate summaries, the quality of the semantic search, and the overall cost of running the service.

- **Transcription Accuracy**: We tested the system with a set of audio clips in both English and Hindi and measured a Word Error Rate (WER) of under 5%. That translates to about **95% accuracy**. The results were a bit worse for clips that had a lot of background noise, which is expected.
- **Summarization Latency**: We measured the time from when the transcription was ready to when we got the summary and tags back from the AI. The 95th percentile latency was **1.8 seconds** with GPT-4o-mini, which feels nearly instant to the user.
- **Semantic Recall Quality**: This was harder to measure, but we created a test set of 100 notes with known connections. We checked if the top 3 most relevant notes appeared in the top 5 suggestions from our semantic search. We hit a **precision@5 of 0.82**, which means our suggestions are highly relevant.
- **Cost**: The running costs are mostly from API calls to AWS Transcribe and OpenAI. A key finding was that using GPT-4o-mini instead of the full GPT-4 model cut our AI processing costs by about **80%**, without a noticeable drop in the quality of the summaries or tags.

VII. Lessons Learned and Design Trade-offs

Building Brainer taught us a lot. Here are some of the biggest takeaways.

- **Model Selection is Everything**: It’s a constant balancing act between an AI model’s power, its speed, and its cost. We learned that for specific tasks like summarization, a smaller, faster model often gives you the best bang for your buck. The biggest model isn’t always the right answer.
- **Keep it Stateless**: In a serverless architecture, stateless solutions are your friend. Using polling instead of WebSockets made our system much simpler and more resilient, and the performance trade-off was worth it.
- **Prompt Engineering is a Process**: Getting a reliable JSON output from the LLM took a lot of trial and error. You have to be incredibly specific in your prompt about the format you want.
- **Embeddings Need Care**: Generating and storing vector embeddings is the easy part. The hard part is managing them. You need a solid background process for creating new embeddings when notes are edited to keep everything in sync.

VIII. Future Work

We’re just getting started. Here’s where we see Brainer going next:

1) **Deeper AI Integration**: We want to go beyond just summarization and let users ”chat” with their notes. This will mean building a full Retrieval-Augmented Generation (RAG) pipeline.
2) **Collaboration**: We want to add features that let users share notes and build a shared knowledge base with others.
3) **Offline Mode**: We plan to use PWA technologies to make the app work offline, maybe even running a smaller version of the AI model directly in the browser.

IX. Conclusion

Brainer is our attempt to show what’s possible when you thoughtfully combine modern AI with cloud services. We’ve built a note-taking system that’s more than just a passive place to store information; it’s an active partner in helping you manage your knowledge. The mix of multilingual voice input, automatic processing, and semantic search creates a uniquely fluid experience. We believe the architectural choices we made, especially using serverless and managed AI services, offer a solid and cost-effective blueprint for anyone looking to build the next wave of intelligent apps.

References

[1] A. Vaswani et al., ”Attention is all you need,” in NIPS, 2017.
[2] T. Brown et al., ”Language models are few-shot learners,” in NeurIPS, 2020.
[3] Next.js 14 Documentation, Vercel Inc., 2023. [Online]. Available: https://nextjs.org/docs
[4] AWS Transcribe Developer Guide, Amazon Web Services, 2023. [Online]. Available: https://docs.aws.amazon.com/transcribe/
[5] OpenAI API Documentation, OpenAI, 2023. [Online]. Available: https://beta.openai.com/docs/
[6] Prisma Documentation, Prisma, 2023. [Online]. Available: https://www.prisma.io/docs/
[7] pgvector: Vector similarity search for Postgres. [Online]. Available: https://github.com/pgvector/pgvector