

JULY 15, 2026 / #NODE.JS

How to Build a RAG Chatbot for Your Docs with Node.js, Google Gemini, and pgvector



Zia Ullah

I was helping a team that had a 200-page API documentation PDF. Every new engineer spent their first two weeks Ctrl+F-ing through it, asking the same questions in Slack, getting redirected to the same paragraphs on page 47.

The doc was accurate. It was even well-written. But nobody could find anything in it fast enough for it to be useful.

That's the problem RAG, or Retrieval-Augmented Generation, solves.

The naïve approach is to stuff your entire PDF into a prompt and let the model figure it out. That breaks down fast: context windows overflow, costs spike on every request, and the model loses the thread somewhere in the wall of text.

RAG takes a different approach. Your documents get broken into small chunks upfront. Ask it a question and it digs out the 3 or 4 chunks that best match it — those are what the model actually sees. The model gets a tight, focused context. The answer comes from what your document actually says — not from whatever the LLM memorized during training.

In this tutorial, you'll build that from scratch. Upload any PDF — an API reference, an internal spec, a research paper — and ask questions about it in plain English. The system finds the relevant sections and answers from the document itself, not from general training data.

The stack: Node.js with Express, Google Gemini for embeddings, Groq for text generation, and pgvector running in Docker. Every

piece of it is free — no credit card, no trial period.

The complete code is on GitHub at [nodejs-rag-chatbot](#).

Table of Contents

- [How RAG Works](#)
- [What We're Building](#)
- [Prerequisites](#)
- [Project Setup](#)
- [Set Up Postgres with pgvector Using Docker](#)
- [Connect to the Database](#)
- [Build the Ingestion Pipeline](#)
- [Build the Query Pipeline](#)
- [Build the Chat API with Express](#)
- [Test the Chatbot](#)
- [Troubleshooting](#)
- [How to Swap in OpenAI](#)
- [What to Build Next](#)

How RAG Works

RAG has two phases, and the code maps directly to both.

Ingestion phase — runs once when you upload a document:

1. Pull the raw text out of the PDF
2. Break it into chunks of 400 to 600 characters each, with a bit of overlap so nothing important gets cut at a boundary

3. Run each chunk through an embedding model, which turns it into a vector (a long list of numbers that captures what the text means)
4. Store each chunk and its vector in Postgres

Query phase — runs every time someone asks a question:

1. Embed the user's question using the same model
2. Search the database for chunks whose vectors are closest to the question vector
3. Take the top 5 matching chunks and assemble them into a context block
4. Send context + question to the LLM and return its answer

The reason this works better than keyword search: the embedding model captures *meaning*, not just exact words. If your doc says "terminate the process" and the user asks "how do I stop it?", vector similarity finds that match. Regular string matching doesn't.

One thing that trips people up: you must use the same embedding model at query time as you did at ingestion. The model defines the geometric space those vectors live in. Switch models halfway through and the coordinates stop meaning the same thing — you'd be comparing apples to completely different apples.

What We're Building

The architecture is intentionally minimal: two endpoints, with nothing you don't need:

- `POST /ingest` : accepts a PDF upload, chunks it, embeds each chunk, stores vectors in pgvector

- `POST /chat` : accepts a question, retrieves the most relevant chunks, returns an LLM-generated answer

The full tech stack:

- **Node.js + Express** — API layer
- **Google Gemini free API** — `gemini-embedding-001` for embeddings (3,072 dimensions per chunk)
- **Groq free API** — `llama-3.1-8b-instant` for text generation
- **PostgreSQL + pgvector** — vector storage and cosine similarity search, running in Docker
- **pdf-parse** — extracts raw text from PDF buffers

Gemini handles embeddings and Groq handles generation. Splitting them across two providers isn't arbitrary. Gemini's generation API has a quota limit of zero in certain regions (including Pakistan), while Groq works everywhere with no restrictions. Using Groq for generation means this tutorial runs the same way regardless of where you are.

Prerequisites

Before you start:

- Node.js 20+ installed on your machine
- Docker Desktop running (this is how we'll run Postgres locally)
- A free Google Gemini API key (for embeddings)
- A free Groq API key (for text generation)

How to Get Your Free Gemini API Key

1. Go to aistudio.google.com/app/apikey and sign in with a Google account
2. Click "Create API key"
3. Select "Create API key in new project"
4. Copy the key — it starts with `AIzaSy...`

No credit card or billing required.

How to Get Your Free Groq API Key

1. Go to console.groq.com and sign up with Google
2. Click "API Keys" in the left sidebar
3. Click "Create API Key", give it a name, copy the key — it starts with `gsk_...`

Groq is free with generous rate limits and works in all regions.

Project Setup

Create the project directory and initialize it:

```
mkdir nodejs-rag-chatbot
cd nodejs-rag-chatbot
npm init -y
```

Install dependencies:

```
npm install express pg pdf-parse uuid dotenv multer
npm install --save-dev nodemon
```

A quick note on the packages: `multer` is what makes file uploads work on the `/ingest` endpoint. Without it, Express can't parse multipart form data.

`pdf-parse` does the heavy lifting on PDFs, though watch out for scanned PDFs. Those are just images with no text layer underneath, so you'll get back an empty string.

`pg` talks to Postgres, `uuid` gives each row a unique ID, and `dotenv` loads your keys before the app does anything.

Create a `.env` in the project root. It needs seven values:

```
GEMINI_API_KEY=AIzaSy...      ← your Gemini key from Google A
GROQ_API_KEY=gsk_...          ← your Groq key from console.gr
POSTGRES_USER=rag_user
POSTGRES_PASSWORD=rag_pass     ← choose any password, this is
POSTGRES_DB=rag_db
DATABASE_URL=postgresql://rag_user:rag_pass@localhost:5432/rag_db
PORT=3000
```

One thing: the password in `POSTGRES_PASSWORD` and the one in `DATABASE_URL` must match exactly. I changed just one of them once and spent way too long debugging a "password authentication failed" error before realising the two values were out of sync.

Update `package.json` scripts:

```
"scripts": {
  "start": "node src/index.js",
  "dev": "nodemon src/index.js"
}
```

Create the `src` directory:

```
mkdir src
```

Your final folder structure will look like this:

```
nodejs-rag-chatbot/
├── src/
│   ├── index.js      ← Express app entry point
│   ├── db.js         ← Postgres connection and schema setup
│   ├── embeddings.js ← Gemini embedding + Groq generation
│   ├── ingest.js     ← Document ingestion pipeline
│   └── query.js      ← RAG query pipeline
├── docker-compose.yml
├── .env
└── package.json
```

Set Up Postgres with pgvector Using Docker

`pgvector` adds a `vector` column type to Postgres and the operators needed to search it by similarity. Normally you'd have to install it yourself, but the `pgvector/pgvector` Docker image ships with it already baked in. Just pull the image and you're good.

Now add `docker-compose.yml` to the project root:

```
services:
  postgres:
    image: pgvector/pgvector:pg16
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
```

```
    POSTGRES_DB: ${POSTGRES_DB}
  ports:
    - "5432:5432"
  volumes:
    - pgdata:/var/lib/postgresql/data

volumes:
  pgdata:
```

Those `${VARIABLE}` references get swapped out from `.env` when Compose starts — so `docker-compose.yml` itself stays clean. This is worth doing from day one. I've seen people skip this and regret it after a repo goes public.

Start it:

```
docker compose up -d
```

Connect to the Database

Create `src/db.js`. This sets up the connection pool and creates the `documents` table on first run:

```
const { Pool } = require('pg');

const pool = new Pool({
  connectionString: process.env.DATABASE_URL,
});

async function initDb() {
  await pool.query(`CREATE EXTENSION IF NOT EXISTS vector`);

  await pool.query(`
    CREATE TABLE IF NOT EXISTS documents (
      id UUID PRIMARY KEY,
      content TEXT NOT NULL,
```

```

        source TEXT NOT NULL,
        embedding VECTOR(3072)
    )
`);

console.log('Database ready');
}

module.exports = { pool, initDb };

```

The `VECTOR(3072)` dimension matches the output of Gemini's `gemini-embedding-001` model exactly. If you use a different embedding model in the future, check its output dimensions and update this number to match.

Build the Ingestion Pipeline

Start with `embeddings.js`. This file is the bridge to both external APIs — Gemini for turning text into vectors, Groq for generating the final answer. Keeping both in one place means a single file to touch if you ever swap providers.

src/embeddings.js:

```

const GEMINI_KEY = process.env.GEMINI_API_KEY;
const GEMINI_BASE = 'https://generativelanguage.googleapis.com/v1'

async function embedText(text) {
  const res = await fetch(
    `${GEMINI_BASE}/gemini-embedding-001:embedContent?key=${GEMINI_KEY}`
  ) {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ content: { parts: [{ text }]} })),
  }
};

const data = await res.json();
if (!res.ok) throw new Error(JSON.stringify(data));
return data.embedding.values;

```

```

}

async function generateAnswer(context, question) {
  const res = await fetch(
    'https://api.groq.com/openai/v1/chat/completions',
    {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
        'Authorization': `Bearer ${process.env.GROQ_API_KEY}`,
      },
      body: JSON.stringify({
        model: 'llama-3.1-8b-instant',
        messages: [
          {
            role: 'system',
            content: 'You are a helpful assistant. Answer the question',
          },
          {
            role: 'user',
            content: `Context:\n${context}\n\nQuestion: ${question}`,
          },
        ],
      }),
    },
  );
  const data = await res.json();
  if (!res.ok) throw new Error(JSON.stringify(data));
  return data.choices[0].message.content;
}

module.exports = { embedText, generateAnswer };

```

We're calling both APIs directly with Node.js's built-in `fetch` rather than the official SDKs. The reason is practical: Google's Node.js SDK routes requests through the `v1beta` endpoint by default, and `gemini-embedding-001` isn't available there — only on `v1`. Direct `fetch` sidesteps that entirely and keeps the dependency count low.

src/ingest.js:

```

const pdfParse = require('pdf-parse');
const { v4: uuidv4 } = require('uuid');
const { pool } = require('./db');
const { embedText } = require('./embeddings');

function chunkText(text, chunkSize = 500, overlap = 50) {
  const chunks = [];
  let start = 0;

  while (start < text.length) {
    const end = Math.min(start + chunkSize, text.length);
    chunks.push(text.slice(start, end).trim());
    start += chunkSize - overlap;
  }

  return chunks.filter(chunk => chunk.length > 50);
}

async function ingestDocument(buffer, filename) {
  const { text } = await pdfParse(buffer);
  const chunks = chunkText(text);

  console.log(`Processing ${chunks.length} chunks from "${filename}"`);

  for (const chunk of chunks) {
    const embedding = await embedText(chunk);

    await pool.query(
      `INSERT INTO documents (id, content, source, embedding)
      VALUES ($1, $2, $3, $4::vector)`,
      [uuidv4(), chunk, filename, JSON.stringify(embedding)]
    );
  }

  return chunks.length;
}

module.exports = { ingestDocument };

```

500 characters per chunk, with 50 characters of overlap between neighbours.

Why the overlap? Without it, a sentence that straddles a boundary gets split, half in one chunk, half in the next — and neither piece

makes sense on its own when retrieved. The overlap keeps those boundary sentences intact.

For most technical docs, 500 is a good starting point. Dense legal or financial text tends to need something closer to 300.

Build the Query Pipeline

src/query.js:

```
const { pool } = require('./db');
const { embedText, generateAnswer } = require('./embeddings');

async function queryDocuments(question) {
  const questionEmbedding = await embedText(question);

  const { rows } = await pool.query(
    `SELECT content, source,
       1 - (embedding <=> $1::vector) AS similarity
    FROM documents
    ORDER BY embedding <=> $1::vector
    LIMIT 5`,
    [JSON.stringify(questionEmbedding)]
  );

  if (rows.length === 0) {
    return { answer: 'No relevant documents found.', sources: [] }
  }

  const context = rows.map(r => r.content).join('\n\n---\n\n');
  const answer = await generateAnswer(context, question);

  return {
    answer,
    sources: [...new Set(rows.map(r => r.source))],
    topSimilarity: parseFloat(rows[0].similarity).toFixed(3),
  };
}

module.exports = { queryDocuments };
```

The `<=>` operator is pgvector's cosine distance. Semantically similar text produces vectors that point in the same direction — so the distance between them is small. Flip that with `1 - distance` and you get a similarity score, where anything close to 1 means a strong match.

I found 0.7 to be a reliable threshold in my testing — chunks above that were almost always relevant. Anything below 0.5 and the retrieval was really stretching, pulling chunks that shared a keyword or two but weren't actually answering the question.

When that happens, the system prompt instruction ("if the context does not contain enough information, say so clearly") becomes important. A well-behaved model will tell the user it doesn't know rather than guess.

We also surface the source filename. Once you've ingested more than one document, users need to know whether that answer came from the architecture spec or the incident report.

Build the Chat API with Express

src/index.js:

```
require('dotenv').config();
const express = require('express');
const multer = require('multer');
const { initDb } = require('./db');
const { ingestDocument } = require('./ingest');
const { queryDocuments } = require('./query');

const app = express();
const upload = multer({ storage: multer.memoryStorage() });

app.use(express.json());

app.post('/ingest', upload.single('file'), async (req, res) => {
```

```

if (!req.file) {
  return res.status(400).json({ error: 'No file uploaded' });
}

if (!req.file.mimetype.includes('pdf')) {
  return res.status(400).json({ error: 'Only PDF files are supported' });
}

try {
  const count = await ingestDocument(req.file.buffer, req.file.mimetype);
  res.json({ message: `Ingested ${count} chunks from "${req.file.mimetype}"` });
} catch (err) {
  console.error(err);
  res.status(500).json({ error: 'Ingestion failed', detail: err.message });
});

app.post('/chat', async (req, res) => {
  const { question } = req.body;

  if (!question || typeof question !== 'string') {
    return res.status(400).json({ error: 'question is required' });
  }

  try {
    const result = await queryDocuments(question);
    res.json(result);
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Query failed', detail: err.message });
  }
});

const PORT = process.env.PORT || 3000;

initDb().then(() => {
  app.listen(PORT, () => {
    console.log(`RAG chatbot running on port ${PORT}`);
  });
});

```

`memoryStorage()` keeps the uploaded file in a buffer instead of writing it to disk. We parse it and store the chunks immediately, so there's nothing to save.

Test the Chatbot

Start the server:

```
npm run dev
```

You should see:

```
Database ready  
RAG chatbot running on port 3000
```

Upload a PDF. Any PDF works. I tested with a copy of a Node.js best practices guide:

```
# Linux / macOS  
curl -X POST http://localhost:3000/ingest -F "file=@your-document.pdf"  
  
# Windows PowerShell  
curl.exe -X POST http://localhost:3000/ingest -F "file=@your-document.pdf"
```

Response:

```
{ "message": "Ingested 142 chunks from \"your-document.pdf\"" }
```

Now ask a question:

```
# Linux / macOS
curl -X POST http://localhost:3000/chat \
  -H "Content-Type: application/json" \
  -d '{ "question": "How should I handle errors in async function" }'

# Windows PowerShell
curl.exe -X POST http://localhost:3000/chat -H "Content-Type: application/json" -d '{ "question": "How should I handle errors in async function" }'
```

Response:

```
{
  "answer": "For async functions in Node.js, wrap your logic in a try-catch block.",
  "sources": ["your-document.pdf"],
  "topSimilarity": "0.841"
}
```

The `topSimilarity` score tells you how well the retrieval went. Above 0.7 and the chunks pulled were genuinely relevant. Below 0.5, and the search was struggling: it found something, but not something that actually answers the question.

Try asking about something your PDF doesn't mention. If the system prompt is doing its job, the model should say it doesn't have enough information rather than making something up. That's the behaviour you want in production.

The repo includes two diagnostic scripts that are useful if anything isn't working:

- `node test-keys.js` — tests both API keys live and reports whether each one succeeds
- `node list-models.js` — fetches the full list of Gemini models available to your API key

Run these before diving into the troubleshooting section below.

Troubleshooting

Everything in this section is a real error I hit while building this. Nothing hypothetical.

Port 5432 is already in use

```
Error: bind: address already in use
```

Something else — probably a local Postgres install — is already on that port. Two fixes are needed. First, in `docker-compose.yml`:

```
ports:
  - "5433:5432"
```

Second, update `DATABASE_URL` in `.env`:

```
DATABASE_URL=postgresql://rag_user:rag_pass@localhost:5433/rag_db
```

The container itself still listens on 5432 internally. You're just changing which port your machine uses to reach it.

Password authentication failed for user "rag_user"

```
Error: password authentication failed for user "rag_user"
```

The password Postgres was initialized with doesn't match what your app is sending. Open `.env` and compare `POSTGRES_PASSWORD` with the password embedded in `DATABASE_URL`. They need to be character-for-character identical.

After fixing the mismatch, the old volume still has the wrong password baked into it. You must destroy it and start fresh:

```
docker compose down -v
docker compose up -d
```

The `-v` flag deletes the data volume. Postgres reinitializes on the next start with the credentials from your current `.env`.

Gemini model not found (404)

```
{ "error": { "code": 404, "message": "models/text-embedding-004 :
```

The Google AI model naming has changed. Older tutorials and blog posts reference model names that no longer exist on the v1 endpoint. The correct model for this stack is `gemini-embedding-001`. That's what this repo uses.

If you want to see every model available to your API key, run:

```
node list-models.js
```

That script fetches the live list directly from the API so you're not guessing.

Vector dimension mismatch

```
ERROR: expected 768 dimensions, not 3072
```

This error appears when your database table was created with a different dimension count than what your embedding model produces. `gemini-embedding-001` outputs 3,072-dimensional vectors. The `documents` table in this tutorial uses `VECTOR(3072)` to match.

If you get this error, it means either an old table exists with the wrong dimension, or you changed embedding models without recreating the table. Drop the data volume and restart:

```
docker compose down -v  
docker compose up -d
```

Vector index dimension limit

```
ERROR: ivfflat index type only supports up to 2000 dimensions
```

pgvector's `ivfflat` and `hnsw` index types have a maximum dimension of 2000. Since `gemini-embedding-001` produces 3,072-dimensional vectors, neither index type works.

This tutorial drops the index and lets pgvector do a full scan — fine for development and any reasonably sized corpus. Scaling to thousands of documents in production? Pick a model under 2000 dimensions. OpenAI's `text-embedding-3-small` outputs 1536 and plays nicely with both index types.

Port 3000 is already in use

```
Error: EADDRINUSE: address already in use :::3000
```

Some other process got there first. Swap the port number in `.env`:

```
PORT=3002
```

Save it and restart the server.

Gemini generation returns quota exceeded (limit: 0)

```
{ "error": { "status": "RESOURCE_EXHAUSTED", "message": "Quota ex
```

That `limit 0` means Google has switched off free generation in your country entirely — not that you've used it up. I hit this myself while testing from Pakistan.

That's exactly why this tutorial uses Groq instead. Make sure `GROQ_API_KEY` is in your `.env` and that `generateAnswer` in `src/embeddings.js` is pointing at `api.groq.com`.

To verify both keys work, run:

```
node test-keys.js
```

It tests the Gemini embedding endpoint and the Groq generation endpoint independently and reports whether each succeeds.

nodemon doesn't pick up changes to .env

nodemon only watches .js files — .env changes don't trigger a restart. Switch to the terminal running the server and type `rs`, then hit Enter. That forces a restart and picks up whatever you changed.

curl doesn't work in Windows PowerShell

```
curl : The term 'curl' is not recognized
```

or

```
curl : Cannot bind parameter because parameter 'Method' is speci
```

PowerShell has a built-in `curl` alias that points to `Invoke-WebRequest` — completely different flags, completely different behaviour. Add `.exe` and you bypass the alias and hit the real binary.

So instead of `curl`, type `curl.exe`:

```
# Ingest
curl.exe -X POST http://localhost:3000/ingest -F "file=@your-docu

# Chat
curl.exe -X POST http://localhost:3000/chat -H "Content-Type: app
```

That .exe is the whole fix.

Docker Desktop stopped running

Docker Desktop doesn't start automatically after a reboot on most setups. If your Docker commands suddenly fail with connection errors, that's probably why. Open Docker Desktop, wait until it says "Engine running", then try again.

How to Swap in OpenAI

If you want to use the OpenAI API instead of Gemini, it's three changes.

1. Install the OpenAI SDK:

```
npm install openai
```

2. Replace src/embeddings.js entirely:

```
const OpenAI = require('openai');

const client = new OpenAI({ apiKey: process.env.OPENAI_API_KEY });

async function embedText(text) {
  const result = await client.embeddings.create({
    model: 'text-embedding-3-small',
```

```

        input: text,
    });
    return result.data[0].embedding;
}

async function generateAnswer(context, question) {
    const result = await client.chat.completions.create({
        model: 'gpt-4o-mini',
        messages: [
            { role: 'system', content: 'Answer only from the context provided' },
            { role: 'user', content: `Context:\n${context}\n\nQuestion: ${question}` }
        ],
    });
    return result.choices[0].message.content;
}

module.exports = { embedText, generateAnswer };

```

3. Update the vector dimension in `src/db.js`:

Open `db.js` and swap `VECTOR(3072)` for `VECTOR(1536)` — that's the output size of `text-embedding-3-small`. Then kill the volume so the table gets recreated with the right dimensions:

```

docker compose down -v
docker compose up -d

```

Nothing else needs touching. The ingestion and query logic works the same regardless of which model you plugged in.

What to Build Next

What you've built works. But there are some gaps that come up quickly once you put it in front of real users.

The most noticeable one is **streaming**. Right now `/chat` holds the connection open until Groq finishes generating the full answer, then returns everything at once. On a short question that's fine. On a longer one, the user stares at nothing for a few seconds and wonders if the request hung.

The Groq API supports streaming — add `stream: true` to the request body and tokens start coming back as they're generated. Piping those through Express with `res.write()` is maybe 15 minutes of work and the difference in feel is immediate.

Metadata filtering is the second thing you'll want. Once you've loaded more than a few documents, queries bleed across everything: ask about the API spec and you'll get chunks from the onboarding guide too.

The fix is a `metadata JSONB` column where you store the document ID on ingest, then add `WHERE metadata->>'doc_id' = $1` to the similarity query. Expose it as an optional body field on `/chat`: `{ "question": "...", "docId": "api-spec-v2" }`. Users get scoped results, and you get much cleaner answers.

When your corpus grows into the hundreds of documents, look at **re-ranking**. Vector similarity retrieval is fast but approximate — it finds chunks that are semantically close to the question, not necessarily the ones that most directly answer it.

The pattern is: retrieve the top 20 by cosine distance, then run a cross-encoder over them to re-score by actual relevance, then take the best 5 from that second pass. LangChain.js has a cross-encoder wrapper if you don't want to implement it yourself.

The last thing most people forget until they actually need it is **document management** — the ability to list what's ingested, delete

a specific file, and re-ingest an updated version.

A `DELETE FROM documents WHERE source = $1` handles the delete case. Add a `GET /documents` endpoint that queries `SELECT DISTINCT source FROM documents` and you have a complete enough API for real use.

RAG isn't magic. It's a well-scoped retrieval problem combined with a language model that's been told to stay within its lane.

The quality of your answers depends on three things: how cleanly your PDFs parse, how well your chunk size fits the content type, and how clearly your system prompt instructs the model to say "I don't know" rather than guess. Get those right and you've built something genuinely useful: the kind of thing that saves a new engineer's first two weeks.



Zia Ullah

Zia Ullah is a Full Stack Developer at ValueAdd Solution Scandinavia AB - <https://www.valueadd.se/> in Sweden, with a Master's in Software Engineering and a Bachelor's in Computer Science. He has 13+ years of experience building and deploying cloud applications on Azure, designing CI/CD pipelines with GitHub Actions and Azure DevOps, and implementing secure authentication, authorization, and API security. He is a technical writer whose articles have been published on leading developer platforms and high-authority technology publications, where he shares practical insights on web development, cloud computing, DevOps, cybersecurity, and software engineering. Connect with him on LinkedIn - <https://www.linkedin.com/in/zia-ullah/>

ADVERTISEMENT