DocuMind AI
Technical Documentation

A Retrieval-Augmented Generation System
for Personal Document Intelligence

Author: Mouadi Yassine

Academic Year: 2025–2026

Computer Science Project
University of The National School of Applied Sciences, Kenitra

Contents

1. Introduction

1.1. The Problem with Document Search

Traditional information retrieval systems are often limited by the constraints of keyword matching, which requires users to recall precise terminology to locate relevant data. While shortcuts like "Ctrl+F" are effective for string searches, they lack the ability to synthesize information or bridge the gap between divergent vocabularies. Consequently, navigating dense academic or professional documentation becomes a fragmented process that relies more on linguistic luck than substantive inquiry.

To address this, semantic search prioritizes intent and context over literal syntax. For example, a student might ask, "What are the consequences of missing the final paper deadline?" while the text may only include "late submission penalties", "integrity policites", etc.

By understanding the conceptual relationship between these terms, semantic search enables users to query complex datasets naturally, ensuring that critical insights are accessible even when the specific terminology is unfamiliar.

1.2. Why Retrieval-Augmented Generation (RAG) Works

Retrieval-Augmented Generation (RAG) offers a more elegant yet simple alternative way to LLMs. Instead of building a giant AI that memorizes everything, which has the obvious inconveniences of being slow and expensve, RAG enables an AI to consult external, verified data sources before generating a response. This methodology functions much like a librarian who:

- You ask a question
- The system searches through your documents for relevant sections
- It sends those sections to the AI along with your question
- The AI answers based only on what it found

This means the AI can't make things up, it cites your actual documents, and it doesn't need retraining when you add new files.

2. Technology Stack

2.1. Frontend: React + TypeScript

Why React? Because components within it are like LEGO bricks. Each piece - Navbar, DocumentUpload, ChatInterface - is isolated, testable, and reusable. When something

breaks, we know exactly which brick to fix. The virtual Document Object Model (DOM) means the UI stays fast even when documents load.

Why TypeScript? TypeScript catches errors at compile time, not runtime. When the backend expects `user_id: number` and the user accidentally sends a string, TypeScript immediately tells us where the error is. This saves hours of debugging, if we were to rely on JavaScript.

2.2. Backend: FastAPI

Why FastAPI and not Flask/Django? The answer is simple: Async. When 10 users upload documents simultaneously, Flask handles them one-by-one. FastAPI handles them concurrently. The difference is 10 seconds vs 2 seconds for batch operations.

2.3. Database: PostgreSQL + PGVector

Why one database instead of two? Simplicity. PGVector lets us store vectors right next to the document metadata. When we delete a document, the cascade removes both the metadata *and* its vectors automatically. This simplifies things a lot.

Why PostgreSQL specifically? PostgreSQL is distinguished by its maturity and unwavering reliability. Its resilience is most evident during system failures. In the event of a power loss or mid-process interruption, PostgreSQL utilizes a Write-Ahead Logging (WAL) mechanism to recover and maintain consistency

2.4. AI: Sentence Transformers + Google Gemini

Why Sentence Transformers? They're locally runnable, free, and produce embeddings good for our case. The `all-MiniLM-L6-v2` model gives us 384-dimensional vectors that capture semantic meaning without being massive.

Why Google Gemini? It's currently the best quality/cost ratio for the question-answering part.

3. Frontend Architecture

3.1. Folder Structure

The frontend follows a clear separation:

```
frontend/
 src/
    components/          # The reusable UI pieces part of code
```

```
      Navbar.tsx
      DocumentUpload.tsx
      DocumentList.tsx
      ChatInterface.tsx
   pages/              # Full page components
      dashboard
         DashboardPage.tsx
      login
         LoginPage.tsx
      register
         RegisterPage.tsx
   App.tsx            # Our root component
   main.tsx          # The react bootstrap
```

Each component has one job. `Navbar` just shows the user and logout button. `DocumentUpload` handles file selection and upload. This makes debugging trivial.

3.2. State Management: Lifted State in App.tsx

In this case, we're not gonna use libraries like Redux nor Context API because it'd be overkill for this scale: we wouldn't need 'global state' data that would be needed by different components at once. Instead, I lift state up to `App.tsx`; I shared the states common to their parent:

```
// App.tsx - Central state management
export default function App() {
  const [isLoggedIn, setIsLoggedIn] = useState(false);
  const [username, setUsername] = useState('');
  const [userId, setUserId] = useState(0);

  // This state is passed down to all children
  return (
    <div className="app-container">
      {isLoggedIn ? (
        <DashboardPage
          username={username}
          userId={userId}
          onLogout={handleLogout}
        />
      ) : (
        <LoginPage onSuccess={handleAuthSuccess} />
      )}
    </div>
  );
}
```

The `App.tsx` knows about authentication, then passes everything down as props. Each child component only gets what it needs.

3.3. UI Behavior During Async Operations

The UI never freezes; when the user uploads a document:

1. Button says "Uploading..." and disables
2. File input becomes disabled
3. The user can still navigate, chat, delete other documents
4. Progress could be added with axios interceptors

The key is `setUploading(true)` happening immediately, before the actual upload starts. This gives instant feedback.

3.4. File Upload Mechanism

The upload uses `FormData`, not JSON:

```
const formData = new FormData();
formData.append('file', file);
formData.append('user_id', userId.toString());

await axios.post('http://localhost:8000/documents/upload', formData, {
  headers: { 'Content-Type': 'multipart/form-data' }
});
```

This is crucial because:

- Files can be large
- FormData streams the file. It doesn't load it all into memory
- The browser shows native progress for FormData
- It works with any file type without encoding issues

4. Backend Architecture

4.1. Entry Point: main.py

The `main.py` file is the front door. Every request goes here first:

```
app = FastAPI(
    title="DocuMind AI API",
    description="Made by Yassine",
    version="1.0.0"
)
```

```
6
7  # Global error handling
8  @app.exception_handler(Exception)
9  async def global_exception_handler(request: Request, exc: Exception):
10     print(f"Uncaught exception: {exc}")
11     return JSONResponse(status_code=500, content={"detail": "Internal
           error"})
12
13 # CORS middleware
14 app.add_middleware(
15     CORSMiddleware,
16     allow_origins=["http://localhost:5173"],
17     allow_credentials=True,
18     allow_methods=["*"],
19     allow_headers=["*"],
20 )
21
22 # Include routers
23 app.include_router(auth.router, prefix="/auth", tags=["Authentication"])
24 app.include_router(documents.router, prefix="/documents", tags=["
       Documents"])
```

Notice the CORS configuration: it only allows the frontend origin. If there were a real world project, this would be a real domain.

4.2. Router Separation:

Routers are like departments in a company:

- `auth.py` handles login/register
- `documents.py` handles file operations
- Each has its own responsibility

It simplifies things: If authentication breaks, We look in `auth.py`. If file upload breaks, We look in `documents.py`.

4.3. Dependency Injection: get_db()

FastAPI's dependency injection:

```
1  def get_db():
2      db = SessionLocal()
3      try:
4          yield db
5      finally:
6          db.close()
7
8  @router.post("/upload")
9  async def upload_document(db: Session = Depends(get_db)):
```

```
    # db is automatically provided
    # automatically closed after function
```

Every route gets a fresh database session of its own. If the route crashes, the session closes automatically.

5. Authentication System

5.1. auth.py as the Receptionist

The `auth.py` file is more like a the building receptionist:

- Register first checks in with a "Can I get a keycard?"
- The login checks: "Here's my keycard, let me in". It verifies it and lets the user in.
- The logout is simply ending the session with a simple "I'm leaving, bye".

It's simple at this version. No email verification, no password reset, no OAuth.

Those are important, but (hopefully) they'd be in a v2 of this project,

5.2. Password Hashing with bcrypt

Passwords are *never* stored plaintext:

```
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

def hash_password(password: str):
    return pwd_context.hash(password)  # Adds salt automatically

def verify_password(plain_password: str, hashed_password: str):
    return pwd_context.verify(plain_password, hashed_password)
```

Bcrypt simplifies password security by integrating salting and hashing into a single, automated process. It generates a unique, random salt for every user to prevent rainbow table attacks, ensuring identical passwords result in different hashes. The algorithm employs a "work factor" to perform multiple iterations, making brute-force attempts computationally expensive. To protect against timing attacks, it performs comparisons in constant time, preventing attackers from deducing secrets based on response speeds. Consequently, developers can implement high-level cryptographic security without managing low-level primitives manually.

5.3. Session Management

In this project, we used `withCredentials: true` in axios calls. This sends cookies automatically. The backend doesn't do JWT tokens because:

1. Cookies are simpler
2. They're automatically sent with every request
3. They have expiration built-in
4. The browser handles storage

For this version of this project, it is sufficient.

6. Database Design

6.1. The Database Defense

Data validation happens at three layers:

Layer	Technology	Example
Frontend	TypeScript	`userId: number` type
API	Pydantic	`constr(min_length=3)`
Database	SQL Constraints	`CHECK(LENGTH(username) > 0)`

If a malicious request somehow bypasses TypeScript and Pydantic, the database constraint catches it.

6.2. Entities Of Our Database:

Table	Purpose	Critical Fields
`users`	Store user accounts	`id, username, password_hash`
`documents`	Document metadata	`id, user_id, file_name, uploaded_at`
`document_chunks`	Text chunks + vectors	`id, document_id, user_id, content, embedding`

6.3. Foreign Keys with ON DELETE CASCADE

This is the most important design decision:

```python
# In models.py
user_id = Column(Integer, ForeignKey("users.id", ondelete="CASCADE"))
document_id = Column(Integer, ForeignKey("documents.id", ondelete="
    CASCADE"))
```

When a user is deleted:

1. All their documents are deleted (cascade from users to documents)
2. All document chunks are deleted (cascade from documents to chunks)
3. No orphaned data
4. No manual cleanup code

The database handles all the integrity for us.

6.4. Data Isolation Per User

Every query includes `user_id`:

```python
# Always filter by user_id
chunks = db.query(DocumentChunk).filter(
    DocumentChunk.user_id == user_id
).all()
```

User A never sees User B's documents, even if they guess the document ID.

7. AI & Document Processing Pipeline

7.1. From PDF Bytes to Text

When you upload a PDF, here's what happens:

```python
def read_pdf(file_bytes):
    doc = fitz.open(stream=file_bytes, filetype="pdf")
    text = ""
    for page in doc:
        page_text = page.get_text()
        text += page_text + "\n"
    return text.strip()
```

`fitz` (the PyMuPDF library) reads the PDF directly from bytes, no temporary files. It handles:

- Encrypted PDFs, but with errors
- Scanned PDFs, but rather poorly, considering it's OCR territory)
- Multi-column layouts, rather adequately

7.2. Chunking Logic: The 500-Character Window

Why 500 characters? Too small (100 chars): we lose the context. Too large (1000 chars): embeddings get rather noisy. 500 is the char number we chose.

```
chunk_size = 500
chunks = []

for i in range(0, len(text), chunk_size):
    chunk_text = text[i:i+chunk_size]
    chunks.append(chunk_text)
```

Important: No overlap between chunks. Overlap helps only in doubling the storage.

7.3. Embedding Generation: Why 384 Dimensions

The `all-MiniLM-L6-v2` model produces 384-dimensional vectors. Each dimension represents some abstract feature of the text. Math example:

- "apple fruit" → [0.2, 0.8, -0.1, ..., 0.4]
- "orange citrus" → [0.3, 0.7, -0.2, ..., 0.3]
- "car vehicle" → [-0.8, 0.1, 0.9, ..., -0.5]

Vector embeddings transform words into multi-dimensional coordinates, allowing AI to interpret semantic relationships as mathematical distance. Words with shared contexts, such as "apple" and "orange," are mapped within similar spatial vectors, resulting in a low cosine distance (a measure that calculates the distance of high-dimensional vectors).

Conversely, unrelated terms like "apple" and "car" are positioned far apart in this vector space, resulting in a high distance. By calculating the angle between these vectors, the AI identifies topical relevance based purely on a distance value.

7.4. Singleton Pattern for Model Loading

Loading the embedding model takes: 2 seconds and 400MB RAM. We can't do that per request:

```
_embedding_model = None  # Global variable

def get_embedding_model():
    global _embedding_model
    if _embedding_model is None:
        _embedding_model = SentenceTransformer('all-MiniLM-L6-v2')
    return _embedding_model
```

We utilize the Singleton Pattern: We ensure the model is instantiated only once and persisted in memory for all subsequent operations.

We then load the SentenceTransformer into a global variable; however, for the next thousand requests, we simply return its existing reference.

7.5. Semantic Search with Cosine Distance

Finding relevant chunks uses vector math:

```
# Convert question to vector
question_vector = model.encode(question)

# PostgreSQL finds closest vectors
results = db.query(DocumentChunk).filter(
    DocumentChunk.user_id == user_id
).order_by(
    DocumentChunk.embedding.cosine_distance(question_vector)
).limit(5).all()
```

The `pgvector` extension optimizes similarity searches by utilizing the Hierarchical Navigable Small World (HNSW) index, which organizes high-dimensional vectors into a multilayered graph. Rather than performing an exhaustive search across all records, the system traverses these layers to rapidly locate the nearest neighbors. This process relies on calculating the **Cosine Distance**, represented by the formula:

$$d = 1 - \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}$$

By navigating this mathematical graph, the database can identify topically relevant content within milliseconds, even when scaling to millions of entries. This ensures that the retrieval process remains efficient without sacrificing the semantic accuracy of the results.

7.6. Question Answering with Gemini

The final step: send context + question to Gemini:

```
prompt = f"""
You␣are␣a␣helpful␣assistant.␣Answer␣based␣strictly␣on␣context.

CONTEXT:
{context_text}

QUESTION:
{question}
"""

response = model.generate_content(prompt)
return response.text
```

We use the "strictly on context" part to prevent Gemini from using its general knowledge, which defeats the purpose of RAG in entirety.

8. API Routes in Detail

8.1. POST /documents/upload

1. Validate file type (PDF, TXT, Excel, CSV)
2. Read file content based on type
3. Create document record
4. Split text into 500-character chunks
5. Generate embedding for each chunk
6. Store chunks with vectors
7. Commit transaction

If step 6 fails, the entire transaction rolls back. No half-uploaded documents.

8.2. GET /documents/list/{user_id}

Returns metadata only, not content:

```
[
  {
    "id": 1,
    "file_name": "financial_report.pdf",
    "uploaded_at": "2025-01-15T10:30:00Z"
  }
]
```

Why not content? Privacy and bandwidth. The list might show 50 documents - sending all content would be megabytes.

8.3. DELETE /documents/{document_id}

```
@router.delete("/{document_id}")
def delete_one(document_id: int, db: Session = Depends(get_db)):
    doc = db.query(Document).filter(Document.id == document_id).first()
    if doc:
        db.delete(doc)  # Cascade deletes chunks automatically
        db.commit()
    return {"message": "Deleted"}
```

The `ON DELETE CASCADE` is pretty crucial: we delete the document, PostgreSQL deletes all its chunks.

8.4. DELETE /documents/all/{user_id}

This deletes every document for a user. A confirmation dialog in the frontend is included.

8.5. POST /documents/query

The RAG endpoint:

1. Generate question embedding
2. Find 5 most similar chunks
3. Concatenate as context
4. Send to Gemini
5. Return answer

If no documents exist, we simply return "Upload documents first" instead of crashing.

9. Infrastructure & Deployment

9.1. Docker Compose Orchestration

The `docker-compose.yml` defines three services:

```
services:
  db:
    image: ankane/pgvector:latest  # PostgreSQL + vector extension
    environment:
      POSTGRES_PASSWORD: 2040
    volumes:
      - postgres_data:/var/lib/postgresql/data

  backend:
    build: ./backend
    ports: ["8000:8000"]
    depends_on: [db]

  frontend:
    build: ./frontend
    ports: ["5173:5173"]
    depends_on: [backend]
```

One command: `docker-compose up`, and everything starts.

9.2. Startup Race Conditions

PostgreSQL takes 5-10 seconds to start. The backend would crash if it tried to connect immediately.

We therefore included this solution: The `startup.sh` file:

```
echo "Waiting for database to be ready."
sleep 5  # Simple but effective
python init_db.py
uvicorn app.main:app --host 0.0.0.0 --port 8000
```

9.3. Database Connection Retry Logic

In `database.py`:

```
def create_engine_with_retry(max_retries=10, delay=5):
    for attempt in range(max_retries):
        try:
            engine = create_engine(DATABASE_URL)
            with engine.connect() as conn:
                conn.execute(text("SELECT 1"))
            return engine
        except OperationalError:
            if attempt == max_retries - 1:
                raise
            time.sleep(delay)
```

If the database isn't ready, it waits 5 seconds and try again. This process is repeated 10 times. This handles slow startup and temporary network issues.

9.4. Frontend Docker Optimizations

The frontend Dockerfile has two stages:

1. Build stage: Install dependencies, compile TypeScript
2. Production stage: Copy only compiled files, use nginx

This keeps the production image small: no dev dependencies nor any source code.

10. End-to-End Flow: Upload to Query

10.1. Step 1: User Selects File

Frontend: User clicks "Choose File", selects '`report.pdf`' for example. React then updates the state.

10.2. Step 2: Frontend Uploads

```
const formData = new FormData();
formData.append('file', file);
formData.append('user_id', '123');
axios.post('http://localhost:8000/documents/upload', formData);
```

The browser streams the file.

10.3. Step 3: Backend Receives

FastAPI receives multipart form data. `UploadFile` gives us a file-like object.

10.4. Step 4: Read PDF

`fitz.open(stream=file_bytes)` gives us the advantage of reading directly from memory.

10.5. Step 5: Chunking

2000-character PDF becomes 4 chunks (500 chars each). Each chunk gets an embedding.

10.6. Step 6: Database Insert

Single transaction:

- Insert document record (ID: 456)
- Insert chunk 1 with embedding
- Insert chunk 2 with embedding
- Insert chunk 3 with embedding
- Insert chunk 4 with embedding
- Commit

PostgreSQL executes an automatic Rollback. This ensures that no document record, without its corresponding chunks, is ever persisted. The database simply reverts to its state prior to the start of the transaction

10.7. Step 7: User Asks Question

"What's the summary?" → Frontend sends to `/documents/query`.

10.8. Step 8: Semantic Search

1. Embed question: "What's the summary?" → vector Q
2. Database: find chunks with smallest cosine distance to Q
3. Returns top 5 chunks

10.9. Step 9: Answer Generation

Chunks concatenated as context. Then, it's sent to Gemini with the prompt. Afterwards, the response returns.

10.10. Step 10: Display

Frontend shows answer in chat bubble (tthe frontend). Conversation continues in the same manner.

11. Error Handling & Resilience

11.1. Frontend:

When backend is down:

```
try {
  const response = await axios.get('http://localhost:8000/auth/me');
  handleAuthSuccess(response.data);
} catch (error) {
  // Not "Network Error", just show login screen
  setIsLoggedIn(false);
}
```

No error messages are actually showon. We just show the login form as if the user wasn't logged in.

11.2. Backend: Structured Errors

Every error returns consistent format:

```
{
  "detail": "Could not read PDF",
  "type": "ValueError"
}
```

11.3. Database: Connection Pooling

SQLAlchemy connection pool:

```
engine = create_engine(
    DATABASE_URL,
    pool_size=10,       # 10 persistent connections
    max_overflow=20,    # Can create 20 more if needed
    pool_pre_ping=True # Check connection before use
)
```

This handles 30 concurrent requests without creating 30 database connections.

12. Project Structure Recap

12.1. Root Infrastructure

The project follows a decoupled microservices architecture, managed by Docker Compose. The root directory contains orchestration files and global configuration:

```
.
 .gitignore
 docker-compose.yml
 Doc/
     backend/
     frontend/
     public/
```

12.2. Backend Architecture (FastAPI)

The backend is organized into a modular FastAPI structure, separating the API logic from the data layer:

```
backend/
 app/
    routers/
       auth.py
       documents.py
    database.py
    main.py
    models.py
 init_db.py
 requirements.txt
 startup.sh
```

12.3. Frontend Architecture (React & TypeScript)

The frontend utilizes a feature-based organization. State is "lifted" to `App.tsx` to maintain a clean data flow between the dashboard and its child components:

```
frontend/
 src/
    components/
       ChatInterface.tsx
       DocumentList.tsx
       DocumentUpload.tsx
       Navbar.tsx
    pages/
       dashboard/DashboardPage.tsx
       login/LoginPage.tsx
       register/RegisterPage.tsx
    App.tsx
    main.tsx
    App.css
 vite.config.ts
 Dockerfile
```

13. What Works Well

13.1. The RAG Pipeline

The core functionality - upload, chunk, embed, search, answer - works reliably. For a v1, it's surprisingly effective.

13.2. Docker Compose Setup

One command to run everything. Perfect for development, decent for production.

13.3. Type Safety

TypeScript + Pydantic catches so many errors. The confidence it provides is worth the extra typing.

13.4. Database Design

The three-table schema with cascading deletes is elegant. It handles all use cases without complexity.

14. Intentional Simplifications

14.1. No User Sessions Database

Sessions are stored in FastAPI's memory. This means:

- Restarting backend logs everyone out
- Multiple backend instances don't share sessions

14.2. No File Size Limits

A user could upload a 1GB PDF and crash the system.

14.3. No Rate Limiting

14.4. No Background Processing

Large PDFs block the upload endpoint.

15. Conclusion

DocuMind AI started as a simple question: "Can I make Ctrl+F smarter?" It became a full-stack application with React, FastAPI, PostgreSQL, and vector search.

The system works: You can upload documents, ask questions, get answers based on your documents.

Most importantly, I built every part myself. From the Docker containers to the SQL migrations to the React state management. I understand how it all fits together, where it breaks, and how to fix it.

The code is available on GitHub: `https://github.com/yassinexng/documind`. Feel free to use it, extend it, or learn from it.