

1. Project Overview

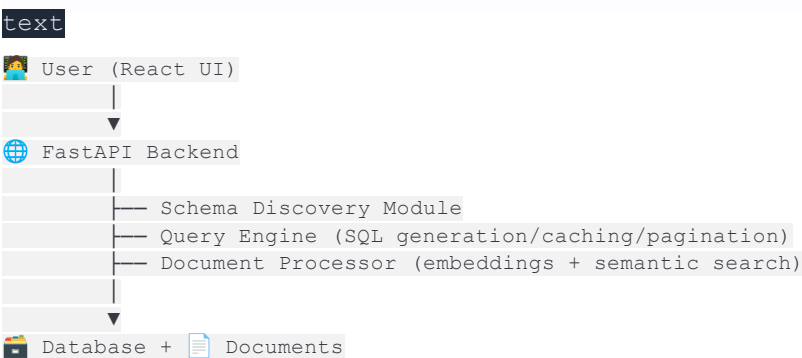
The NLP Query Engine for Employee Data is a full-stack AI Engineering project that allows users to query both structured and unstructured employee information through natural language. The system connects to a database, automatically discovers its schema, processes uploaded documents (PDF/DOCX/TXT/CSV), and interprets natural-language queries to return meaningful results from the correct data source.

Objectives

- Enable database querying without writing SQL.
- Support document content search (resumes, reviews, contracts).
- Combine both (hybrid queries).
- Provide a simple web interface for non-technical users.

2. System Architecture

High-level flow



Technology Stack

Layer	Technologies Used








Frontend	React (HTML, CSS, JavaScript)
Backend (API)	FastAPI (Python 3.8+)
Database	SQLite (test.db)
AI / NLP	Sentence-Transformers (all-MiniLM-L6-v2)
Other Libraries	SQLAlchemy, PyPDF2, python-docx, langchain
Production Utilities	Uvicorn server, CORS middleware, logging, caching

3. Core Components

Component	Purpose / Description
-----------	-----------------------

SchemaDiscovery	Discovers database tables, columns, data types and relationships automatically.
QueryEngine	Classifies queries (SQL / Document / Hybrid), maps keywords to schema, generates and executes SQL. Includes caching, pagination, SQL safety checks, and logging.
DocumentProcessor	Handles document uploads, chunks text, generates embeddings using Sentence Transformers for semantic search and similarity matching.
FastAPI Endpoints	<code>/api/connect-database</code> , <code>/api/upload-documents</code> , <code>/api/query</code> – used by frontend to send requests and view results.
React Frontend	UI with sections for Database Connection, Document Upload, Query Panel, and Results Display.

4. Key Features

-  Dynamic Schema Discovery – no hardcoded tables or columns.
-  Natural Language Querying – plain English → SQL.
-  Document Processing & Semantic Search – uses embeddings for contextual matches.
-  Hybrid Queries – combine results from structured and unstructured data.
-  Performance – caching (TTL 5 min) + pagination for large datasets.
-  Security & Reliability – SQL injection protection and global error handler.
-  Frontend Integration – FastAPI and React communicate via CORS-enabled REST APIs.

5. Setup & Usage

Backend (FastAPI)

`bash`

```
python -m venv venv
venv\Scripts\activate # (Windows)
pip install -r requirements.txt
uvicorn backend.main:app --reload
```

Runs on: <http://127.0.0.1:8000>

Frontend (React)

`bash`

```
cd frontend
npm install
npm start
```

Runs on: <http://localhost:3000>

Running the App – Steps

1. Connect Database:
Enter `sqlite:///backend/test.db` → Click Connect.
Confirms schema discovery.
 2. Upload Documents:
Choose one or more `.txt` files → Click Upload.
Backend processes and creates embeddings.
 3. Ask a Query:
Examples:
 - “How many employees do we have?” (SQL)
 - “Find resumes of Python developers” (Document)
 - “Compare employees in database with resumes” (Hybrid)
 4. View Results:
Database tables render for SQL queries; text snippets display for document queries.
-

6. Example Results

SQL Query:

How many employees do we have?

json

```
{"status": "ok", "query_type": "sql", "results": [[4]]}
```

Document Query:

json

```
{"status": "ok", "query_type": "document", "results": ["Alice Johnson is a Python developer...", "Diana Prince is an HR manager..."]}
```

Hybrid Query:

json

```
{
  "status": "ok",
  "query_type": "hybrid",
  "sql_used": "SELECT emp_id, full_name, salary FROM employees LIMIT 10;",
  "db_results": [[1, "Alice Johnson", 75000.0], [2, "Bob Smith", 90000.0], [3, "Charlie Brown", 60000.0], [4, "Diana Prince", 85000.0]],
  "doc_results": ["Alice Johnson is a Python developer with 5 years of backend experience.", "Diana Prince is an HR manager with expertise in recruitment."]
}
```

Screenshots:

(Add your captured images as described earlier)

1. FastAPI Swagger UI
2. React UI Dashboard
3. Connection Success
4. Query Results

7. Performance & Optimization

Optimization	Description
Caching	Recent query results cached for 5 minutes using TTL cache.

Pagination	Large SQL results split into pages (<code>LIMIT/OFFSET</code>).
Lazy Model Loading	Sentence-Transformer model is loaded only when needed.
Thread Safety	SQLite opened with <code>check_same_thread=False</code> for multi-thread access.
Error Handling	Global exception handler returns JSON errors cleanly.

8. Future Enhancements

- Integrate vector database like FAISS or Chroma for fast document retrieval.
 - Enhance NLP SQL generation with Transformer models (GPT / LLM).
 - Implement authentication and multi-user roles.
 - Enable asynchronous query processing for better scalability.
-

9. Conclusion

The NLP Query Engine intelligently bridges the gap between natural-language users and data sources. It demonstrates auto-schema discovery, language processing, and hybrid data retrieval inside a production-ready FastAPI + React stack. This system showcases practical engineering of AI-integrated applications for real-world enterprise settings.

Author Details



Name: Pooja Marlashikari



Email: poojamarla2001@gmail.com



GitHub Repository: <https://github.com/poojamarla/NLP-Query-Engine>