Smart Bridge – Intelligent SQL Querying

Full Stack Development
Project Documentation

Team ID:	LTVIP2026TMIDS80425
Team Leader:	C Chaithanya Prasad
Team Member:	K Dinesh
Team Member:	Karnam Vidhyasree
Team Member:	Bharath Kumar
Team Member:	Parlapalli Khandith Kumar Reddy

February 2026

Contents

1 Introduction

- **Project Title:** Smart Bridge – Intelligent SQL Querying
- **Team Members and Roles:**

Name	Role	Responsibilities
C Chaithanya Prasad	Team Leader / Backend Lead	Project setup, AI integration, deployment
K Dinesh	Database Engineer	Database schema, seed data, documentation
Karnam Vidhyasree	Frontend Developer	Query panel, results display, UI components
Bharath Kumar	API Developer	RESTful API endpoints, error handling, testing
Parlapalli Khandith Kumar Reddy	UI/UX Developer	Schema viewer, dark theme, responsive layout

2 Project Overview

2.1 Purpose

The purpose of this project is to build an intelligent web application that allows non-technical users to query relational databases using natural language. The system uses Google Gemini AI to translate plain English questions into valid SQL queries, executes them on a connected SQLite database, and presents results in a user-friendly interface. This eliminates the need for SQL knowledge and reduces dependency on IT teams for data retrieval.

2.2 Features

Key features and functionalities of the application:

- **Natural Language Query Input** – Users type questions in plain English
- **AI-Powered SQL Generation** – Google Gemini 1.5 Flash converts NL to SQL
- **Schema-Aware Processing** – AI receives actual database schema for accurate queries
- **Interactive Schema Viewer** – Sidebar displays all tables with column details and data types
- **Formatted Results Display** – Query results shown in clean, sortable tables
- **Generated SQL Display** – Shows the SQL query with syntax highlighting
- **AI Query Explanation** – Natural language explanation of what the query does
- **Premium Dark Theme** – Modern, responsive UI with glassmorphism effects
- **Error Handling** – User-friendly error messages for invalid queries
- **Sample E-Commerce Database** – Pre-loaded data for immediate demonstration

3 Architecture

3.1 Frontend

The frontend is built using **React 18** with **Vite** as the build tool:

- **Component-Based Architecture**: Reusable components for query panel, results display, schema viewer, and header
- **State Management**: React hooks (useState, useEffect) for local state management
- **API Communication**: Fetch API for RESTful communication with the backend
- **Styling**: Custom CSS with CSS variables for theming, glassmorphism effects, and responsive design
- **Build Tool**: Vite for fast HMR (Hot Module Replacement) during development

3.2 Backend

The backend is built using **Python 3.11** with **FastAPI**:

- **FastAPI Framework**: High-performance async web framework with automatic OpenAPI documentation
- **Gemini AI Service**: Dedicated service module for Google Gemini API integration
- **Schema Extraction**: Dynamic database schema reading for AI context
- **Query Execution**: Safe SQL execution with read-only enforcement
- **CORS Middleware**: Cross-Origin Resource Sharing for frontend-backend communication
- **Environment Management**: python-dotenv for secure API key management

3.3 Database

The database uses **SQLite 3**:

- **Schema Design**: 4 related tables forming an e-commerce data model
- **Tables**:
  - `customers` (id, first_name, last_name, email, city, registration_date)
  - `products` (id, name, category, price, stock_quantity)
  - `orders` (id, customer_id, order_date, status, total_amount)
  - `order_items` (id, order_id, product_id, quantity, unit_price)
- **Relationships**: Foreign keys linking orders to customers and order_items to both orders and products
- **Seed Data**: Pre-populated with realistic sample data for demonstration

4 Setup Instructions

4.1 Prerequisites

- Python 3.11 or higher
- Node.js 18 or higher
- npm 9 or higher
- Google Gemini API key (from `https://ai.google.dev/`)
- Git

4.2 Installation

1. **Clone the repository:**

```
git clone <repository-url>
cd SMART_BRIDGE_INTELLIGENT_SQLQUERYING
```

2. **Backend setup:**

```
cd backend
pip install -r requirements.txt
```

3. **Create `.env` file in the backend directory:**

```
GEMINI_API_KEY=your_google_gemini_api_key_here
```

4. **Seed the database:**

```
python seed_db.py
```

5. **Frontend setup:**

```
cd ../frontend
npm install
```

5 Folder Structure

5.1 Client (Frontend)

```
frontend/
            src/
                  components/
                        QueryPanel.jsx    # NL query input
    component
                        ResultsDisplay.jsx # Query results table
                        SchemaViewer.jsx   # Database schema
    sidebar
```

```
                        Header.jsx          # App header component
                App.jsx                   # Main application
    component
                App.css                   # Application styles
                main.jsx                  # React entry point
                index.css                 # Global styles
            index.html                  # HTML entry point
            package.json                # Dependencies & scripts
            vite.config.js              # Vite configuration
```

5.2 Server (Backend)

```
backend/
            main.py                 # FastAPI application & routes
            gemini_service.py       # Google Gemini AI integration
            seed_db.py              # Database seeding script
            requirements.txt        # Python dependencies
            .env                    # Environment variables (API
    keys)
            store.db                # SQLite database file
```

6 Running the Application

- **Backend:**

```
cd backend
uvicorn main:app --reload --port 8000
```

The backend server starts at `http://localhost:8000`

- **Frontend:**

```
cd frontend
npm run dev
```

The frontend dev server starts at `http://localhost:5173`

7 API Documentation

Method	Endpoint	Parameters	Description
GET	/api/schema	None	Returns database schema (tables, columns, types)
POST	/api/query	{"query": "string"}	Accepts NL query, returns generated SQL, results, and explanation
GET	/health	None	Health check endpoint

GET	/docs	None	Auto-generated Swagger API documentation

Example – POST /api/query:

Request:

```
{
    "query": "Show me the top 5 customers by total spending"
}
```

Response:

```
{
    "sql": "SELECT c.first_name, c.last_name,
            SUM(o.total_amount) AS total_spending
            FROM customers c JOIN orders o
            ON c.id = o.customer_id
            GROUP BY c.id ORDER BY total_spending
            DESC LIMIT 5;",
    "results": {
        "columns": ["first_name", "last_name",
                    "total_spending"],
        "rows": [["John", "Doe", 1500.00], ...]
    },
    "explanation": "This query joins the customers
        and orders tables..."
}
```

8 Authentication

The current version of the application does not implement user authentication, as it is designed as a local development tool / demo application. Security is handled through:

- **API Key Management**: Google Gemini API key stored in `.env` file, loaded via `python-dotenv`
- **Read-Only Execution**: Only SELECT queries are permitted; INSERT, UPDATE, DELETE, and DROP statements are blocked
- **Input Sanitization**: User inputs are sanitized before processing
- **CORS Configuration**: Restricted to configured frontend origins

Future Enhancement: JWT-based authentication with role-based access control is planned for production deployment.

9 User Interface

The application features a premium dark-themed interface with the following key UI elements:

- **Header Bar**: Application title and branding with gradient accent
- **Query Panel**: Large text area for natural language input with a submit button featuring hover effects
- **Results Section**: Formatted table with alternating row colors, generated SQL in syntax-highlighted code block, and AI explanation text
- **Schema Sidebar**: Expandable tree view of database tables showing column names and data types
- **Loading States**: Animated loading indicators during AI processing
- **Error States**: Clear error messages with retry options

10 Testing

10.1 Testing Strategy

- **Unit Testing**: Individual function testing for schema extraction, SQL sanitization, and API response formatting
- **Integration Testing**: End-to-end flow testing from NL input to results display
- **API Testing**: Endpoint testing using FastAPI's built-in Swagger UI at `/docs`
- **UI Testing**: Manual testing of all frontend components across browsers
- **Performance Testing**: Response time measurement under various query complexities

11 Screenshots or Demo

1. **Main Interface**: Dark-themed dashboard with query panel and schema viewer
2. **Query Execution**: Results displayed in formatted table after NL query
3. **Schema Browser**: Expanded table view showing columns and data types
4. **Error Handling**: User-friendly error message for invalid queries

(Attach screenshots or provide demo link here)

12 Known Issues

- Very complex multi-table queries with subqueries may occasionally generate incorrect SQL
- The application requires an active internet connection for Google Gemini API access
- Large result sets (1000+ rows) may cause slow rendering in the frontend table
- SQLite-specific SQL syntax may differ from PostgreSQL/MySQL conventions

13 Future Enhancements

- Multi-database engine support (PostgreSQL, MySQL, MongoDB)
- Query history with search and bookmarking
- Auto-generated data visualizations (charts, graphs)
- Voice input for hands-free querying
- User authentication with JWT tokens
- Export results as CSV, Excel, or PDF
- AI-powered query suggestions based on schema
- Docker containerization for easy deployment
- Support for multiple LLM providers (GPT-4, Claude, open-source)
- Real-time collaborative querying

GitHub Link: `https://github.com/Chaithanya182/sql-querying`
Dataset: Sample e-commerce data (embedded in `seed_db.py`)