System Design: AI Chat-to-Visualization

Made by - Kartik Raj, IIT Kanpur

1. Project Overview & Core Architecture

Our objective is to build a real-time, interactive learning application. The system allows a user to ask a conceptual question and receive back both a clear, text-based explanation and a dynamic, animated visualization of that concept.

Our project's AI agent, the **LLM Service**, automates the complex cognitive task of conceptual learning by following a clear **reason, plan, and execute** model.

First, it **reasons** about the user's question (e.g., "Explain Photosynthesis") by deconstructing the core scientific principles involved. Next, it **plans** a two-part, synchronized response: (1) a clear, pedagogical text explanation and (2) a corresponding visual plan, deciding which shapes, colors, and animations will best represent the concept. Finally, it **executes** this plan by generating a single, structured JSON object containing both the `text` string and the detailed `visualization_spec`, which our frontend then renders.

The architecture is a classic decoupled **client-server model** with a real-time data layer.

1. **Frontend (Client):** A **React** single-page application (SPA) that provides the user interface. It is responsible for rendering the chat history, the visualization canvas, and managing the real-time connection.
2. **Backend (Server):** A **Node.js (Express)** API that serves as the system's core. It handles user requests, manages database state, orchestrates the LLM, and broadcasts updates.
3. **LLM Service:** An external or internal service (abstracted) that acts as the "brain." It receives a text prompt and returns a structured JSON object containing the explanation and the visualization specification.
4. **Database:** A **PostgreSQL** database used for persistence, storing all user questions and their corresponding answers.
5. **Real-time Layer: Server-Sent Events (SSE)** are used for unidirectional, real-time communication from the server to all connected clients.

2. Chosen Technologies & Rationale

Component	Technology	Rationale (Why We Chose It)
Backend API	**Node.js (Express)**	**Asynchronous I/O:** Perfect for our use case, which involves waiting for responses from the external LLM API without blocking the server. **SSE Handling:** Node's event-driven nature is highly efficient for managing multiple persistent SSE connections. **Ecosystem:** Fast to prototype and build with using the vast npm ecosystem.
Frontend UI	**React**	**Component-Based:** The UI naturally breaks down into components (ChatPanel, VisualizationCanvas, Controls), which React handles exceptionally well. **State Management:** React's state and context (or hooks) are ideal for managing the "current" visualization, chat history, and playback state.
Real-time	**Server-Sent Events (SSE)**	**Simplicity & Fit:** Unlike WebSockets (which are full-duplex), SSE is one-way (server-to-client). This is exactly our requirement. We only need to *broadcast* updates *from* the server. **Native Support:** EventSource is a built-in browser API, requiring no client-side libraries and offering features like automatic reconnection.
Database	**PostgreSQL**	**Reliability & JSON Support:** We need a reliable, structured database. PostgreSQL provides this, *and* it has excellent native JSONB support, which is perfect for storing the arbitrary visualization JSON specs efficiently.
LLM Orchestration	**Abstracted Service**	**Flexibility:** By wrapping the LLM call in our own internal LLMService, we can swap out the provider (OpenAI, Ollama, etc.) at any time without changing our core API logic. This module is solely responsible for prompt engineering and response parsing.

3. Data Design (Database Schema)

We will use a simplified, relational model. The JSONB type in PostgreSQL is key for storing the visualization spec without needing a complex, normalized schema for animations and layers.

Table: questions

Stores the initial query from the user.

Column	Type	Constraints	Description
id	TEXT	Primary Key	Unique identifier (e.g., "q_123").
user_id	TEXT	Not Null	Identifier for the user (e.g., "u1").
question_text	TEXT	Not Null	The literal question the user asked.
created_at	TIMESTAMPTZ	Default NOW()	Timestamp of when the question was submitted.

Table: answers

Stores the generated response (text and visualization).

Column	Type	Constraints	Description
id	TEXT	Primary Key	Unique identifier (e.g., "a_456").
question_id	TEXT	Foreign Key (to questions.id), Unique	Links this answer directly to its question (1:1).
explanation_	TEXT	Not Null	The text explanation from the LLM.
visualization_spec	JSONB	Not Null	The complete JSON object for the visualization.
created_at	TIMESTAMPTZ	Default NOW()	Timestamp of when the answer was generated.

4. Component Breakdown (High-Level)

Backend (Node.js)

- **server.js**: Main entry point. Initializes Express app, applies middleware (CORS, json()), and mounts routes.
- **routes/api.routes.js**: Defines the four required endpoints:

- POST /questions
- GET /questions
- GET /answers/:id
- GET /stream

- **services/stream.service.js**: Manages the SSE connections.
  - addClient(res): Adds a new client's response object to an active connections list.
  - broadcast(event, data): Iterates over all active clients and writes the SSE message (event: ... data: ...).
- **services/llm.service.js**: Orchestrates the LLM interaction.
  - generateExplanation(questionText):
    1. Constructs a detailed system prompt (e.g., "You are an educator. Respond with a JSON object...").
    2. Calls the external LLM API.
    3. Parses the response.
    4. **Crucially:** Validates the returned JSON against our expected schema.
- **controllers/main.controller.js**: Handles the business logic for each route.
  - handleNewQuestion:
    5. Saves the question to the questions table.
    6. Calls llm.service.generateExplanation().
    7. Saves the result to the answers table.
    8. Calls stream.service.broadcast() with the new data.
    9. Returns the questionId and answerId to the original POST request.
  - handleGetHistory: Fetches all questions and their associated answer IDs (via a JOIN).
  - handleGetAnswer: Fetches a specific answer by its ID.
  - handleStream: Establishes the SSE connection and adds the client to the stream.service.

Frontend (React)

- **App.js**: Top-level component. Manages layout (left/right panes) and fetches initial data.
- **hooks/useChatStream.js**:
    - Connects to the /api/stream endpoint using the EventSource API.
    - Listens for question_created and answer_created events.
    - Updates the application's global state (e.g., via a Context or Zustand store) with new messages, which causes the UI to re-render.
- **components/ChatContainer.js**:
    - ChatHistory.js: Renders the list of question and answer bubbles.
    - QuestionInput.js: The text input form. On submit, it calls POST /api/questions.
- **components/VisualizationContainer.js**:
    - VisualizationCanvas.js: The core renderer.
        - Receives the visualization_spec JSON as a prop.
        - Uses the HTML5 <canvas> API (or SVG) to draw shapes.
        - Uses requestAnimationFrame to run the animation loop, interpolating properties (like x, y, orbit) based on the current time and the spec's animations array.
    - PlaybackControls.js:
        - Contains "Play" and "Pause" buttons.
        - Manages an isPlaying state and a currentTime state, which are passed as props to the VisualizationCanvas to control the animation.

5. API Flow: Submitting a New Question

This is the most critical flow, as it touches every part of the system.

1. **User:** Types "Explain Photosynthesis" and hits send.
2. **React Frontend:**
    - The QuestionInput component's onSubmit handler is triggered.

- It makes a POST /api/questions request with { userId: 'u1', question: '...' }.
- It optimistically adds the user's question bubble to the ChatHistory.

3. **Node.js Backend (Controller):**
   - Receives the POST request.
   - Inserts the question into the questions table (gets back q_789).
   - (Asynchronously) Calls llm.service.generateExplanation('Explain Photosynthesis').
4. **LLM Service:**
   - Waits for the LLM to process the request and stream back the full JSON response.
   - Parses the JSON.
5. **Node.js Backend (Controller):**
   - (Callback/Promise resumes) Receives the text and visualization JSON from the llm.service.
   - Inserts this data into the answers table (gets back a_321).
   - **Broadcasts:** Calls stream.service.broadcast('answer_created', { ...answer_data... }).
   - **Responds:** Sends the initial POST request its response: { questionId: 'q_789', answerId: 'a_321' }.
6. **All Connected Clients (Frontend):**
   - The useChatStream hook hears the answer_created event.
   - It updates the global state, adding the new answer.
   - The ChatHistory re-renders, showing the new text explanation bubble.
   - The VisualizationContainer re-renders, passing the new visualization_spec to the VisualizationCanvas, which begins rendering the animation.