**System Architecture & Data Flow for AI Study Buddy**

This document outlines the system architecture of the AI Study Buddy application and explains the data flow for its core functionalities.

1. **System Architecture Diagram (ASCII)**

The application employs a client-heavy architecture where the user's browser handles most of the logic. The backend consists of serverless, third-party platforms (Google Cloud and Firebase) for AI processing and user authentication.

**1.1 Component Breakdown:**

* **React Frontend:** All UI components responsible for rendering the user interface.
* **App State & Logic:** The core state management and business logic within App.tsx, handling user interactions and orchestrating API calls.
* **Web Speech API:** A browser-native API for real-time voice-to-text transcription. Data processing happens on-device.
* **Local Storage:** The browser's key-value store, used for persisting user study sessions (text, summary, quiz, etc.).
* **Gemini Service:** A client-side module that prepares and sends requests to the Google Gemini API for all AI-related tasks.
* **Firebase Service:** A client-side module for handling user authentication and profile management with the Firebase platform.
* **Google Cloud Platform:** Hosts the Gemini API. All AI tasks are processed here.
* **Firebase Platform:** Provides authentication services and a Firestore database for storing user profile data.

**2. Data Flow Explanations**

**2.1. User Authentication Flow (Login)**

* **Initiation:** User enters their email and password into the LoginForm component and clicks "Login".
* **Client-Side Validation:** The form performs basic checks (e.g., fields are not empty).
* **API Call:** The handleLogin function in App.tsx calls the signInWithEmailAndPassword function from our firebase.ts service.
* **Firebase Request:** The firebase.ts service makes a secure HTTPS request to the Firebase Authentication service, sending the user's credentials.
* **Validation & Response:** Firebase validates the credentials against its user database.
* **On Success:** It returns a JSON Web Token (JWT) and user profile information to the client.
* **On Failure:** It returns an error code (e.g., auth/wrong-password).
* **State Update:** The onAuthStateChanged listener in App.tsx detects the successful login, updates the user state with the returned user object.
* **UI Render:** React re-renders the application, replacing the AuthPage with the main App dashboard.

**2.2. Content Processing Flow (File Upload)**

* **Initiation:** A user drags and drops a PDF or image file into the FileUpload component.
* **File Reading:** The browser's FileReader API reads the file from the user's disk into memory as a Base64 encoded string. No data has left the browser yet.
* **Service Call:** handleFileProcess in App.tsx is triggered, calling extractTextFromFile from geminiService.ts with the Base64 string and the file's MIME type.
* **API Request:** geminiService constructs an HTTPS request to the Google Gemini API. The request payload includes the Base64 data and a prompt asking the model to extract text.
* **AI Processing:** The Gemini API receives the request, decodes the file data, and uses its multimodal capabilities to extract the text content.
* **API Response:** The API sends the extracted text back to the client in the response body.
* **State Update:** The returned text is used to update the sourceText state in App.tsx.
* **UI Render:** The UI updates to reflect that content is ready, enabling the "Summarize," "Generate Quiz," and other action buttons.

**2.3. AI Action Flow (Quiz Generation)**

* **Initiation:** With sourceText available, the user clicks the "Generate Quiz" button in the ActionsPanel.
* **Service Call:** handleAction in App.tsx calls the generateQuiz function in geminiService.ts, passing the sourceText as context.
* **API Request:** The service constructs a request to the Gemini API. The prompt specifically instructs the model to create a 5-question multiple-choice quiz and to return the output in a structured JSON format, adhering to a predefined schema (responseSchema).
* **AI Processing:** The Gemini model analyzes the text and generates the quiz content according to the prompt's instructions, formatting its internal response as JSON.
* **API Response:** The API returns a JSON string in the response body.
* **Data Parsing:** geminiService receives the response and uses JSON.parse() to convert the JSON string into a JavaScript array of QuizQuestion objects.
* **State Update:** The parsed array is used to set the quiz state in App.tsx, and activeOutput is set to 'quiz'.
* **UI Render:** The OutputArea component now displays the QuizView component, which renders the interactive quiz.

**2.4. Session Management Flow (Save Session)**

* **Initiation:** The user clicks the "Save Session" button.
* **State Snapshot:** The handleConfirmSave function in App.tsx reads the current values of sourceText, summary, quiz, and chatHistory from the application's state.
* **Local Storage Read:** The function accesses the browser's localStorage to get the existing array of saved sessions, if any.
* **Data Manipulation:** It creates a new StudySession object containing the state snapshot and a user-provided name. This new object is added to the front of the sessions array.
* **Local Storage Write:** The entire updated sessions array is converted to a JSON string using JSON.stringify() and written back to localStorage under the key studySessions. All data remains within the user's browser.

**3. System Architecture Diagram**

