Developer Brief: Interactive Client Project Dashboard Application

1. Project Title: ScienceDesigner Client Project Dashboard

2. Introduction & Overview:
ScienceDesigner currently tracks client project progress, deliverables, comments, and deadlines using spreadsheets (referencing the previously provided example image). This project aims to replace the spreadsheet system with a dedicated, interactive, cloud-based web application. The application will serve as a central hub for both internal ScienceDesigner staff (Admins) and their clients to view and manage project status in real-time.

3. Project Goals:

- Replace the current spreadsheet-based tracking system.
- Provide a secure, user-friendly interface for both admins and clients.
- Improve efficiency in project tracking and communication.
- Enhance the client experience by providing direct access to project status and history.
- Centralize project information, comments, and related links/files.
- Ensure the platform is scalable to accommodate a growing number of clients and projects.

4. Target Audience:

- **Admins (ScienceDesigner Staff):** Require full access to create, manage, and update all client projects. Need visibility across all clients and projects. Will manage users and clients.
- **Clients (End Users):** Require read-only (or potentially limited write access, e.g., adding comments) access *only* to their specific projects. Need a simple way to log in and view the status, history, and comments related to their work with ScienceDesigner.

5. Key Features (Functional Requirements):

- **User Authentication & Roles:**
  - Secure login system for Admins and Clients.
  - Distinct roles (Admin, Client) with different permissions.
- **Admin Dashboard/Functionality:**
  - View a list of all clients and their associated projects.
  - Create, Read, Update, Delete (CRUD) projects.
  - Assign projects to specific clients.
  - Update project fields (similar to spreadsheet columns: Description, Status, Client Delivery Date, Comments, Resource, Due Date, Links). Status should likely be a predefined dropdown (e.g., "In Progress", "For Review",

"Complete").
    - Add/edit comments on projects (timestamped).
    - Manage users (invite/add clients, manage admin accounts).
    - Search and filter projects (by client, status, date range, etc.).
    - Ability to upload/link relevant files or resources (expanding on the current "Links" column).
- **Client Dashboard/Functionality:**
    - Upon login, view a dashboard listing *only* their associated projects.
    - View project details (Description, Status, Delivery Date, Due Date).
    - View project comments and history.
    - (Optional - To Discuss) Ability for clients to add comments or approve stages.
    - View linked resources/files.
- **Data Representation:**
    - Display project information clearly, likely in a table format similar to the spreadsheet, but with enhanced UI/UX (e.g., color-coding for status).
    - Handle and display dates effectively.
    - Render comments in a readable thread or list.

6. Non-Functional Requirements:

- **Cloud Hosting:** The application must be hosted on a reliable cloud platform (e.g., AWS, Google Cloud, Azure).
- **Security:** Implement standard web security practices (secure authentication, data encryption, protection against common vulnerabilities). Role-based access control is crucial.
- **Scalability:** The architecture should handle growth in users, clients, and projects without significant performance degradation.
- **Usability:** The interface should be intuitive and easy to navigate for both admin and client users.
- **Maintainability:** Code should be well-documented and follow best practices for ease of future updates.
- **Performance:** The application should load quickly and respond promptly to user interactions.

7. Data Model (Initial Thoughts):

- **Users:** (ID, Name, Email, Password Hash, Role [Admin/Client])
- **Clients:** (ID, Company Name, Contact Person, Contact Email, associated User ID) - *Note: A User with a 'Client' role might be directly linked to a Client entity.*
- **Projects:** (ID, Project Number, Client ID, Description, Status, Client Delivery Date, Due Date, Assigned Resource [User ID or Name], Creation Date, Last Updated

Date)
- **Comments:** (ID, Project ID, User ID, Comment Text, Timestamp)
- **Resources/Links:** (ID, Project ID, Link URL, File URL [if uploads needed], Description, Timestamp)

8. Existing Assets:

- The previously provided spreadsheet image (image_83ef85.jpg) serves as the primary reference for data fields and initial functionality.

9. Deliverables:

- Fully functional, deployed web application meeting the specified requirements.
- Complete source code repository.
- Documentation covering setup, deployment, and basic usage/administration.

10. Contact Person:

- [Your Name/Project Manager's Name]
- [Your Email/Contact Info]

Software/Technology Stack Suggestions:

Choosing the right stack depends on factors like existing team expertise (if any), budget, and specific feature nuances. Here are a few common and robust options for building this type of cloud-based application:

Option 1: Python Backend (Django/Flask) + JavaScript Frontend (React/Vue/HTMX)

- **Backend:**
  - **Django:** A high-level Python framework that encourages rapid development. Its built-in admin interface is excellent for quickly building the admin management features. Very robust and scalable.
  - **Flask:** A more lightweight Python microframework. Offers more flexibility but requires adding more components (like an ORM, admin interface) separately.
- **Frontend:**
  - **React / Vue / Angular:** Powerful JavaScript libraries/frameworks for building interactive user interfaces. Ideal for a smooth, modern user experience.
  - **HTMX / Server-Side Templates:** If interactivity needs are simpler, using server-rendered templates (Django/Flask templates) perhaps enhanced with HTMX can reduce frontend complexity.
- **Database:** PostgreSQL or MySQL (Relational databases are well-suited for this structured data).
- **Cloud:** AWS (EC2/RDS, Elastic Beanstalk), Google Cloud (App Engine, Cloud

SQL), Azure (App Service, Azure SQL Database), Heroku.
- **Pros:** Python is widely used, great libraries, Django admin is a huge plus for admin features. Mature ecosystem.
- **Cons:** Can involve managing separate frontend/backend codebases if using React/Vue.

Option 2: Full-Stack JavaScript (Node.js Backend + React/Vue/Angular Frontend)

- **Backend:**
  - **Node.js with Express/Koa/NestJS:** Using JavaScript on the backend allows for language consistency across the stack. Node.js is performant for I/O-bound applications. NestJS provides more structure, similar to Django/Spring.
- **Frontend:**
  - **React / Vue / Angular:** Same as Option 1. These integrate very naturally with Node.js backends.
- **Database:** PostgreSQL, MySQL, or MongoDB (NoSQL could be an option, though relational might be a better fit here).
- **Cloud:** Same as Option 1. Also platforms like Vercel (frontend) and Supabase/Firebase (Backend-as-a-Service options).
- **Pros:** Single language (JavaScript) for front and back end, large npm ecosystem, good for real-time features (with WebSockets).
- **Cons:** Can be less opinionated than Django, potentially requiring more setup decisions.

Option 3: Ruby on Rails

- **Backend/Frontend:** Rails is a full-stack framework using Ruby. It follows convention over configuration, enabling rapid development. Handles backend logic and frontend views (often using its templating system or integrating with JS frameworks).
- **Database:** PostgreSQL or MySQL are common choices.
- **Cloud:** Heroku (historically very popular for Rails), AWS, Google Cloud, Azure.
- **Pros:** Very high developer productivity, strong conventions, mature ecosystem.
- **Cons:** Ruby is less common than Python/JavaScript in some areas, smaller talent pool potentially.

Recommendation Considerations:

- **For Rapid Development & Strong Admin Interface: Python/Django** is often a top contender due to its built-in admin.
- **For Full JavaScript Stack: Node.js + React/Vue** is excellent if you prefer

JavaScript end-to-end or anticipate needing highly dynamic, real-time UI features.

- **Database: PostgreSQL** is generally a highly recommended, robust, open-source relational database suitable for most web applications, including this one.
- **Cloud: AWS, Google Cloud, or Azure** are the big three, offering a wide range of services and scalability. Choose based on potential existing infrastructure, team familiarity, or specific service pricing/features. Heroku can be simpler for initial deployment but may become more expensive as you scale.

Discuss these options with your developer or development team based on their expertise and your long-term vision for the application.