Design Choices

1. **Functional Components with React Hooks:**
    - I opted for functional components with React Hooks (`useState`, `useEffect`, etc.) to maintain a clean and modern design. Functional components simplify the code and eliminate the need for class components, while hooks enable me to manage state and lifecycle methods effectively.
    - Using hooks like `useState` for local state and `useEffect` for fetching images when the component mounts provides a clean, declarative way to manage component lifecycles.
2. **State Management:**
    - The primary state management in the app is handled through local state with `useState`. Each component manages its own state and passes down necessary data and functions as props. This makes each component focused and responsible for its own part of the app.
    - In `ImageUpload`, multiple pieces of state are tracked, such as the upload progress, file preview, and drag-and-drop status. Managing all this state locally within the component keeps things scoped and avoids unnecessary re-renders across the entire app.
3. **Pagination in ImageGallery:**
    - Pagination was implemented to avoid overwhelming the UI when a large number of images are present. I chose a simple client-side pagination using `slice()` to break up the images into manageable pages. While this works fine for small datasets, I planned for scalability by suggesting Firestore server-side pagination using limits and cursors.
    - The pagination is controlled by `currentPage`, `totalPages`, and functions like `handleNextPage` and `handlePreviousPage` to enable easy navigation between pages.
4. **Firebase Integration:**
    - The app uses Firebase for both Firestore (for storing metadata like image URLs and upload dates) and Firebase Storage (for storing the actual image files). This setup provides a cloud-based solution with minimal configuration, suitable for real-time updates and simple authentication, should I need to add user accounts.
    - Image uploading uses `uploadBytesResumable` to track the progress of the upload, while `getDownloadURL` retrieves the image URL once the upload is complete. This ensures a responsive user experience, providing real-time feedback during the upload process.
5. **CSS for Consistency and Design:**
    - I used modular CSS for styling the components (`Navbar.css`, `ImageUpload.css`, `ImageGallery.css`) to maintain separation between logic and presentation. This approach helps keep the styling concerns isolated, and each component has its own scoped styles, avoiding conflicts.

    - The app also includes custom vectors and images for a visually appealing UI (such as the background image and icons in buttons like delete and view). This attention to design enhances the user experience and makes the app more engaging.
6. **Conditional Rendering for Feedback:**
    - Conditional rendering was used to handle different upload states in `ImageUpload` (e.g., uploading, upload complete, or idle). This provides feedback to the user on the current process and keeps the UI dynamic based on the user's actions.

Challenges Faced

1. **Firebase Storage and Firestore Synchronization:**
    - One challenge was synchronizing image uploads in Firebase Storage with Firestore. The upload process needed to ensure that after the image was uploaded to Firebase Storage, the Firestore database was updated with the image URL and metadata. This required handling the asynchronous nature of both Firebase services and ensuring the state was updated correctly.
2. **Handling Large Datasets in Pagination:**
    - While client-side pagination (`slice()` method) works well for small datasets, I faced the challenge of scalability. If the app stores a large number of images, the client-side approach can slow down performance. I considered moving to server-side pagination with Firestore, but this was not implemented in the current version due to time constraints.
3. **Handling File Size and Progress During Upload:**
    - Managing the real-time upload progress and file size display presented some complexity. I used Firebase's `uploadBytesResumable` method to track the upload progress and state to show the user how much of the file had been uploaded. Ensuring that the progress bar and file size were calculated accurately while keeping the app responsive was tricky, especially for large image files.

Features Implemented:

- Drag-and-Drop Image Upload
- Image Deletion with Confirmation
- Image Modal and Full-View Display
- Real-time Feedback for Uploads

Explanation for Not Using Child Components in `ImageGallery`

In `ImageGallery`, I did not create child components for the following reasons:

1. **No Immediate Reusability Needs:**
   - The design of the app does not currently reuse image-related functionalities (like displaying or deleting images) outside of the `ImageGallery` component. Since the gallery is the only place where these actions are needed, breaking it down into smaller components would add unnecessary complexity without any clear benefit at this point.
   - For example, a child component could be created for each image in the grid, but since no other part of the app needs this logic, this would increase the number of components without adding significant value.