Technology Justification

1. Choice of Framework: Next.js

Next.js was selected as the framework for this project due to its full-stack capabilities, which make it a highly pragmatic choice for building MVPs that require both frontend interactivity and backend data handling. Its combination of server-side rendering (SSR), static generation (SSG), API routes, and modern React features allows a single framework to handle both the client-facing UI and backend logic, simplifying development, deployment, and maintenance.

Specifically, Next.js provides several key advantages for this interactive project:

1. **Fullstack Capability**: The ability to create API routes alongside frontend pages reduces the need for a separate backend service, streamlining the architecture for a small but interactive MVP. This allowed me to colocate the data-fetching logic, validation, and endpoint responses directly within the project structure.

2. **Parallel and Incremental Routing**: Next.js supports parallel routing, which enables independent loading and error states for different sections of the page. For example, the lens selector and scene preview components can fetch and render data asynchronously without blocking each other. This improves perceived performance and enhances the user experience.

3. **Server-Side Rendering (SSR)**: SSR allows the initial page to render quickly with pre-fetched data, improving the Largest Contentful Paint (LCP) and SEO performance. This is particularly important for image-heavy interactive content, where a slow first paint could negatively impact user engagement.

4. **Image Optimization**: Out-of-the-box image optimization automatically serves responsive images and optimizes them for different devices and viewport sizes.

For a project that displays high-resolution lens and scene images, this feature significantly reduces load times and improves performance.

5. **React Ecosystem Benefits**: Next.js leverages modern React features such as hooks and context, enabling clean state management and highly composable UI components. This supports the complex interaction patterns required for the lens and scene selection workflow.

In summary, Next.js was chosen because it **enables a performant, full-stack solution** that reduces architectural overhead, supports modern frontend interactivity, and ensures maintainable, scalable code. Its built-in optimizations and flexible routing model make it a natural fit for this task, where the user experience relies on fast, reactive updates in response to lens and scene selections.

2. Backend Architecture and Design Decisions

The backend for this project was designed to handle **on-demand, minimal data retrieval** while maintaining strong validation and reliability. Two JSON data sources were provided: one for products (lenses) and another for scenes. The design goals were to avoid overwhelming the client with unnecessary data, ensure robust validation, and provide a clean API contract.

2.1 Data Handling and Endpoint Design

1. Incremental Data Loading:

Rather than serving a single, large dataset containing all 58 lenses with their 4 associated scenes each (Mountain, Beach, Road, and Naked eye view), the backend exposes **targeted endpoints** to fetch only the data required by the client at any given time:

- `/lenses`: Returns the list of all lenses (`id`, `name`, `sku`). This powers the lens selector.

- `/lenses/:sku/scenes/:name`: Returns a single scene for a given lens sku and scene type (the name of the scene). This powers the scene preview component.

This approach reduces payload size, improves perceived performance, and ensures the client only processes what is necessary for the current interaction.

2. Data Modeling and Mapping:

The `buildLensSceneMap` utility acts as an **adapter function**, mapping lens SKUs to their respective scene objects. This simplifies data access, enables O(1) retrieval per lens, and decouples the frontend from the raw JSON structure. Scenes are modeled with a naked-eye image and a scene-specific image, allowing for a direct comparison without redundant data transfer.

3. Validation Separation:

Validation schemas are deliberately separated from the models and stored in `src/utils/validations` instead of alongside `src/models`. This separation enforces **single responsibility** for each folder:

- `src/models` focuses purely on the shape and structure of domain entities.
- `src/utils/validations` handles input validation and parsing for incoming request data or query parameters.

This ensures consistent validation across the project while keeping models clean and reusable.

4. Robust Error Handling:

All incoming requests are validated, and missing or malformed data triggers backend errors without exposing sensitive information. This protects the client from rendering inconsistencies and provides a clear separation of concerns between server errors and user input errors.

5. Endpoint Naming Conventions:

Endpoints are named to be **predictable, RESTful, and descriptive**:

- `/lenses` clearly returns all lenses.
- `/lenses/:sku/scenes/:name` explicitly communicates that it retrieves a specific scene for a given lens.

This convention improves maintainability and allows the frontend to construct requests dynamically based on user selections.

3. Frontend Architecture and Design Decisions

The frontend was built to maximize interactivity while maintaining performance, leveraging **Next.js features** and modern React patterns.

3.1. Parallel Routing for Independent Loading States:

By using Next.js parallel routes, the lens/scene selector and scene preview components can load independently. If one fetch fails or takes longer, it does not block the rest of the page, providing a smoother user experience.

3.2. State Management via URL Parameters:

Lens and scene selections are fully represented in the URL (`sku` and `sceneType`). This allows for:

- Bookmarking and sharing specific views.
- Automatic hydration of the state when refreshing the page or navigating directly via URL.

Using `useSearchParams` ensures client components reactively update when URL parameters change.

3.3 Data Fetching with SWR:

SWR was chosen for its **stale-while-revalidate strategy**, providing:

- Automatic caching to avoid redundant requests.
- Optimistic UI updates.
- Simplified error and loading state management.

For example, the lens list and scene data are fetched via SWR, allowing the UI to remain responsive while data is being retrieved or updated.

4. Interactive Scene Preview:

The scene comparison component uses a **draggable divider** for side-by-side image comparison. Images leverage Next.js Image optimization with the `priority` attribute for above-the-fold content, improving Largest Contentful Paint (LCP) and overall UX performance.

5. Validation and Error States on the Frontend:

Query parameters are validated using the separated schemas before triggering data fetches. Invalid or missing parameters render an `EmptySelection` component rather than breaking the page, ensuring robust UX even with unexpected inputs.

4. Summary

By choosing Next.js as the framework and carefully structuring the backend and frontend:

- The project remains full-stack within a single framework, reducing overhead.
- The backend serves small, on-demand datasets, validated and mapped for efficient access.
- The frontend is highly interactive, leveraging parallel routes, SWR caching, and reactive URL-based state.

This architecture allows for a performant, maintainable, and user-friendly MVP that scales logically as additional lenses or scenes are added.

5. Future Improvements / Time Constraints

Given more time, additional enhancements could have been implemented to further improve maintainability, reliability, and scalability:

1. Automated Testing:

- **Unit Tests:** For utility functions such as `buildLensSceneMap` and API client methods, ensuring data transformations and error handling remain correct.
- **Integration Tests:** For API endpoints to verify correct responses for valid and invalid inputs.
- **Component Tests:** For React components like the lens/scene selector, and `ImageCompare`, validating UI behavior and state updates.

2. Enhanced Caching Strategies:

- Implement server-side caching for `buildLensSceneMap`, lenses, and scene data to reduce repeated computations on frequently accessed data.

3. **Accessibility Improvements:**

- Ensure all interactive elements (like the scene comparison divider) have clear focus states and ARIA attributes.

Including these would increase confidence in code quality and reduce regression risk, while maintaining a high-quality user experience.