Development Standards

4.1 Coding Standards

4.1.1 Naming Convention

▪ Give meaningful name to variables, functions, components, and classes.
▪ Use camelCase for defining variable and function names. Camel case refers to words having the first letter of an identifier is lowercase and the first letter of each subsequent concatenated words is capitalized. Example: `handleSubmit`, `getUserData`.
▪ Use PascalCase for React component names and TypeScript interfaces/types. Pascal Case is one in which the first letter in the identifier and the first letter of each subsequent concatenated word are capitalized. Example: `UserProfile.tsx`, `FarmerDashboard.tsx`.
▪ Use UPPER_SNAKE_CASE for constants. Example: `MAX_UPLOAD_SIZE`, `API_ENDPOINTS`.
▪ Prefix boolean variables with "is", "has", or "should". Example: `isLoading`, `hasPermission`.
▪ Use plural names for arrays. Example: `users`, `cropImages`.

4.1.2 Organization of Files

▪ Follow the Next.js App Router structure for pages and components.
▪ Group related functionality in feature-based directories.
▪ We should follow a modified MVC architecture adapted for React/Next.js:

- Models: Data interfaces and API handlers in `src/lib`
- Views: React components in `src/components`
- Controllers: Business logic in custom hooks and context providers

▪ There should be a dedicated section for app-wide constants:

- All API endpoints should be defined in `src/constants/api.ts`
- UI constants like dimensions should be in `src/constants/ui.ts`
- Feature flags should be in `src/constants/features.ts`

▪ All reusable utility functions should be placed in `src/utils` directory.
▪ All images, videos, and static assets should be in the `public` directory with appropriate subdirectories.
▪ All CSS modules should be co-located with their respective components.

4.1.3 Formatting and Indentation

The KrishiSagar project follows a consistent code style to make it easy for developers to work with the codebase. We use Prettier and ESLint to enforce these standards.

4.1.3.1 Indenting and Whitespace

▪ Use an indent of 2 spaces, with no tabs.
▪ Lines should have no trailing whitespace at the end.
▪ Files should be formatted with \n as the line ending (Unix line endings), not \r\n (Windows line endings).
▪ All text files should end in a single newline (\n).
▪ Maximum line length should be 100 characters.
▪ Use blank lines to separate logical sections of code.

4.1.3.2 TypeScript Specific

▪ Always specify types for function parameters and return values.
▪ Use interfaces for object shapes (not types) unless using union or intersection types.
▪ Avoid using `any` type - use `unknown` when the type is truly not known.
▪ Use type assertions only when necessary and when you can guarantee the type.

4.1.3.3 React Specific

▪ Use functional components with hooks instead of class components.
▪ Component files should export only one component.
▪ Destructure props in function parameters.
▪ Use the React Fragment shorthand (`<>...</>`) when possible.
▪ Name event handlers with the pattern `handle[Event]` . Example: `handleClick` , `handleInputChange` .

4.1.3.4 Operators

▪ All binary operators (operators that come between two values), such as `+` , `-` , `=` , `!=` , `===` , `>` , etc. should have a space before and after the operator, for readability.
▪ For example, an assignment should be formatted as `const foo = bar;` rather than `const foo=bar;` .
▪ Unary operators (operators that operate on only one value), such as `++` , should not have a space between the operator and the variable.

4.1.3.5 Control Structures

▪ Control statements should have one space between the control keyword and opening parenthesis, to distinguish them from function calls.
▪ Always use curly braces even in situations where they are technically optional. Having them

increases readability and decreases the likelihood of logic errors being introduced when new lines are added.

▪ Opening braces should be on the same line as the statement:

```
if (condition) {
  // code
} else {
  // code
}
```

4.1.4 Comments and Documentation

▪ Use JSDoc comments for functions, interfaces, and classes.
▪ Each component should have a brief description of its purpose.
▪ Complex logic should be explained with inline comments.
▪ Use TODO comments for code that needs to be revisited, including the developer's name.
▪ Keep comments up-to-date when changing code.

4.1.5 Import Structure

▪ Group imports in the following order, separated by blank lines:

1. Built-in React and Next.js imports
2. External libraries
3. Internal absolute imports (components, hooks, etc.)
4. Internal relative imports
5. Asset imports (CSS, images, etc.)

4.1.6 State Management

▪ Use local component state with useState for simple, component-specific state.
▪ Use Context API for state that needs to be accessed by multiple components.
▪ Use SWR for remote data fetching and caching.
▪ Avoid prop drilling by utilizing context or custom hooks.

4.1.7 Error Handling

▪ Use try/catch blocks for handling asynchronous errors.
▪ Create custom error types for different kinds of application errors.
▪ Log errors appropriately based on severity.
▪ Provide user-friendly error messages in the UI.

4.1.8 Performance Considerations

- Memoize expensive calculations with useMemo.
- Use React.memo for expensive component renders.
- Optimize re-renders by using useCallback for event handlers passed as props.
- Use virtualization for long lists.
- Implement code splitting using Next.js dynamic imports.