Project Workflow Documentation

Project Structure

```
project-root/
├── src/
│   ├── components/
│   │   ├── Header.jsx
│   │   └── other components...
│   ├── services/
│   │   └── aiService.js
│   ├── models/
│   │   └── AI model classes...
│   ├── styles/
│   │   └── tailwind styles...
│   └── App.jsx
├── public/
│   └── assets/
├── .env
├── package.json
├── vite.config.js
└── tailwind.config.js
```

Technology Stack

Core Technologies

- **React 18+** - Frontend framework
- **Vite** - Build tool and development server
- **Tailwind CSS v4** - Utility-first CSS framework
- **AI Services Integration** - Multiple LLM providers

Development Tools

- **Node.js** - Runtime environment
- **npm/yarn** - Package management
- **Git** - Version control
- **ESLint** - Code linting
- **Prettier** - Code formatting

Development Workflow

1. Setup and Installation

```
# Clone the repository
git clone [repository-url]

# Install dependencies
npm install

# Create environment file
cp .env.example .env

# Start development server
npm run dev
```

2. Component Development

1. **Create Component**

   ```
   // src/components/NewComponent.jsx
   import React from 'react'

   export default function NewComponent() {
     return (
       <div className="container mx-auto px-4">
         {/* Component content */}
       </div>
     )
   }
   ```

2. **Implement Tailwind Styling**

   - Use utility classes
   - Follow responsive design patterns
   - Maintain consistent spacing
3. **Add Component Logic**

   - Implement React hooks
   - Handle state management
   - Add event handlers

3. AI Integration Process

1. **Service Setup**

```
// src/services/aiService.js
import { Configuration, OpenAIApi } from 'openai'

export class AIService {
  constructor() {
    this.initializeProviders()
  }

  async generateResponse(prompt) {
    // Implementation
  }
}
```

2. **Environment Configuration**

```
VITE_OPENAI_API_KEY=your_key
VITE_ANTHROPIC_API_KEY=your_key
VITE_GOOGLE_API_KEY=your_key
VITE_HF_API_KEY=your_key
```

4. Testing and Quality Assurance

1. **Component Testing**

```
import { render, screen } from '@testing-library/react'
import YourComponent from './YourComponent'

test('renders component', () => {
  render(<YourComponent />)
  // Add assertions
})
```

2. **Code Quality**
   - Run linter: `npm run lint`
   - Format code: `npm run format`
   - Review PR checklist

5. Deployment Process

1. **Build Application**

```
npm run build
```

2. **Environment Verification**

    - Check all environment variables
    - Verify API keys and endpoints
    - Test production build locally
3. **Deployment Steps**

    - Push to staging
    - Run automated tests
    - Deploy to production

Git Workflow

1. Branch Strategy

```
# Feature development
git checkout -b feature/new-feature

# Bug fixes
git checkout -b fix/bug-description

# Release preparation
git checkout -b release/v1.0.0
```

2. Commit Convention

```
# Format
type(scope): description

# Examples
feat(auth): add login component
fix(header): correct logo alignment
docs(readme): update installation steps
```

3. Pull Request Process

1. Create PR with description
2. Add screenshots/videos if UI changes
3. Link related issues
4. Request reviews
5. Address feedback
6. Merge after approval

Continuous Integration

1. GitHub Actions Workflow

```
name: CI
on: [push, pull_request]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Install dependencies
        run: npm install
      - name: Run tests
        run: npm test
```

2. Quality Gates

- All tests passing
- Code coverage > 80%
- No linting errors
- Successful build
- PR review approved

Monitoring and Maintenance

1. Performance Monitoring

- Track page load times
- Monitor API response times
- Check resource usage

2. Error Tracking

- Implement error logging
- Monitor API failures
- Track user issues

3. Updates and Maintenance

- Regular dependency updates
- Security patches
- Performance optimizations

Best Practices

1. Code Organization

- Follow component structure
- Maintain service separation

- Use proper naming conventions

2. Performance

- Implement code splitting
- Optimize images and assets
- Use proper caching strategies

3. Security

- Secure API endpoints
- Validate user input
- Protect sensitive data

Documentation

1. Code Documentation

- Add JSDoc comments
- Document complex logic
- Maintain README files

2. API Documentation

- Document endpoints
- Provide usage examples
- Include response formats

3. Deployment Documentation

- Document build process
- Include environment setup
- List deployment steps