# Technical Design Document - Architecture Overview

## Document Information

**Application**: CareIQ Builder **Version**: 0.1.092 **Framework**: ServiceNow UI Core **Last Updated**: 2024 **Authors**: Development Team

## Executive Summary

CareIQ Builder is a ServiceNow-based assessment authoring application that integrates with the CareIQ platform. It enables clinical staff to create, manage, and publish complex healthcare assessments with sophisticated conditional logic, clinical relationships, and care plan integration.

### Key Capabilities

* **Assessment Authoring**: Create multi-section clinical assessments
* **Conditional Logic**: Implement triggered questions based on answer selections
* **Clinical Integration**: Link answers to problems, goals, interventions, and guidelines
* **Version Management**: Maintain multiple versions with draft/published workflows
* **Library Integration**: Reuse pre-validated content from CareIQ library
* **Real-time Validation**: Test assessments before deployment

## System Architecture

### High-Level Architecture

┌─────────────────────────────────────────────────────────────┐  
│ ServiceNow Platform │  
│ ┌────────────────────────────────────────────────────────┐ │  
│ │ CareIQ Builder (UI Component) │ │  
│ │ ┌──────────────────────────────────────────────────┐ │ │  
│ │ │ View Layer (index.js - 21,166 lines) │ │ │  
│ │ │ - Snabbdom Virtual DOM │ │ │  
│ │ │ - Component Rendering │ │ │  
│ │ │ - User Interface │ │ │  
│ │ └──────────────────────────────────────────────────┘ │ │  
│ │ ┌──────────────────────────────────────────────────┐ │ │  
│ │ │ State Management (index.js) │ │ │  
│ │ │ - Initial State (8,500+ lines) │ │ │  
│ │ │ - Action Handlers (100+ actions) │ │ │  
│ │ │ - State Updates │ │ │  
│ │ └──────────────────────────────────────────────────┘ │ │  
│ │ ┌──────────────────────────────────────────────────┐ │ │  
│ │ │ Effects Layer (effects.js) │ │ │  
│ │ │ - HTTP Effects (50+ endpoints) │ │ │  
│ │ │ - API Call Management │ │ │  
│ │ └──────────────────────────────────────────────────┘ │ │  
│ └────────────────────────────────────────────────────────┘ │  
│ ┌────────────────────────────────────────────────────────┐ │  
│ │ ServiceNow Scripted REST APIs (api.js) │ │  
│ │ - Request Handling │ │  
│ │ - Data Transformation │ │  
│ │ - Security Layer │ │  
│ └────────────────────────────────────────────────────────┘ │  
│ ┌────────────────────────────────────────────────────────┐ │  
│ │ CareIQ Services Script Include │ │  
│ │ (x\_1628056\_careiq.CareIQServices) │ │  
│ │ - Backend Communication │ │  
│ │ - Business Logic │ │  
│ └────────────────────────────────────────────────────────┘ │  
└─────────────────────────────────────────────────────────────┘  
 ↓  
 HTTPS / REST  
 ↓  
┌─────────────────────────────────────────────────────────────┐  
│ CareIQ Platform (External) │  
│ - Assessment Storage │  
│ - Clinical Content Library │  
│ - PGI (Problem-Goal-Intervention) Management │  
│ - Guideline Repository │  
└─────────────────────────────────────────────────────────────┘

## Technology Stack

### Frontend

**Primary Framework** - @servicenow/ui-core (v24.1.1) - ServiceNow’s UI component framework - Based on reactive, component-based architecture - Declarative state management

**Rendering** - @servicenow/ui-renderer-snabbdom (v24.1.1) - Virtual DOM rendering - High-performance DOM updates - JSX-like syntax support (with limitations)

**HTTP Communication** - @servicenow/ui-effect-http (v24.1.1) - Effect-based HTTP calls - Declarative API requests - Integrated with action/state system

**Styling** - @servicenow/sass-kit - ServiceNow styling framework - Custom SCSS (styles.scss) - Component-scoped styles

**Build Tools** - Babel runtime (v7.25.7) - JavaScript transpilation - Sass (v1.53.0) - CSS preprocessing - ServiceNow build pipeline

### Backend

**ServiceNow Platform** - Scripted REST API endpoints - Script Includes for business logic - Application scope: x\_cadal\_careiq\_b\_0 (builder scope) - Integration scope: x\_1628056\_careiq (CareIQ services scope)

**External Integration** - CareIQ Platform - External healthcare assessment platform - RESTful API communication - OAuth/token-based authentication

## Component Architecture

### File Structure

CareIQ Builder/  
├── src/  
│ └── cadal-careiq-builder/  
│ ├── index.js # Main component (21,166 lines)  
│ │ ├── View Layer # Lines 1-12,000+  
│ │ ├── State Management # Initial state definition  
│ │ └── Action Handlers # Lines 12,000+  
│ ├── effects.js # HTTP effects definitions  
│ ├── config-actions.js # Configuration actions  
│ ├── core-actions.js # UI state actions  
│ ├── utils.js # Utility functions  
│ └── styles.scss # Component styles  
├── api/  
│ └── [multiple api.js files] # Scripted REST API endpoints  
└── now-ui.json # Component configuration

### Component Responsibilities

#### **index.js - Main Component**

**Lines 1-12,000+: View Layer** - Renders entire UI using snabbdom virtual DOM - Manages component hierarchy - Handles user interactions - Dispatches actions based on events

**Key Sections:** - Assessment list view - Builder interface (sections, questions, answers) - Relationship modals - Confirmation dialogs - System messages - Loading overlays

**Lines 12,000+: State and Action Handlers** - Initial state definition (8,500+ lines of state structure) - 100+ action handlers for all operations - State update logic - Effect dispatching

#### **effects.js - HTTP Effects**

Defines all HTTP effects using createHttpEffect:

export const MAKE\_ASSESSMENT\_DETAILS\_REQUEST = createHttpEffect(  
 '/api/x\_cadal\_careiq\_b\_0/careiq\_builder\_api/get-sections',  
 {  
 method: 'POST',  
 dataParam: 'requestBody',  
 headers: {'Content-Type': 'application/json'},  
 startActionType: 'START',  
 successActionType: 'SUCCESS',  
 errorActionType: 'ERROR'  
 }  
);

**Total Effects**: 50+ HTTP effects covering: - Assessment CRUD operations - Section management - Question/answer operations - Relationship management - PGI operations - Typeahead searches - Configuration loading

#### **config-actions.js**

Configuration and initialization actions: - Load CareIQ configuration - Fetch use case categories - Initialize application state - Handle authentication

#### **core-actions.js**

UI state management: - Mobile view detection - Panel expansion/collapse - System message management - Modal visibility - Loading states

#### **utils.js**

Utility functions: - Data transformation - Formatting helpers - Validation functions - Common calculations

#### **api.js Files (Scripted REST APIs)**

ServiceNow server-side API endpoints: - Receive requests from UI component - Transform data for CareIQ platform - Handle authentication - Manage errors - Return responses

**Pattern:**

(function process(request, response) {  
 var requestData = request.body.data; // Stash immediately  
  
 // Business logic  
 var careiqServices = new x\_1628056\_careiq.CareIQServices();  
 var result = careiqServices.builderAddQuestion(requestData);  
  
 // Safe error handling (no e.message access)  
 try {  
 response.setBody(result);  
 } catch (e) {  
 var errorMsg = 'Server error occurred';  
 if (e && typeof e.toString === 'function') {  
 errorMsg = e.toString();  
 }  
 response.setError(errorMsg);  
 }  
})(request, response);

## Design Principles

### 1. Security-First Architecture

**All external API calls routed through ServiceNow:** - No direct client-side calls to CareIQ platform - ServiceNow acts as secure proxy - Authentication managed server-side - Request validation and sanitization

**Rationale**: Prevents exposing credentials and sensitive endpoints to client

### 2. Declarative State Management

**Single source of truth:** - All application state in one object - State updates through explicit actions - Predictable state transitions - Easy to debug and trace

**Example:**

// State definition  
const state = {  
 assessments: [],  
 currentAssessment: null,  
 selectedSection: null,  
 questionChanges: {},  
 // ... 8,500+ lines of state  
};  
  
// State updates only through actions  
'UPDATE\_QUESTION\_LABEL': (coeffects) => {  
 const {action, updateState} = coeffects;  
 updateState({  
 questionChanges: {  
 ...state.questionChanges,  
 [action.payload.questionId]: {  
 ...state.questionChanges[action.payload.questionId],  
 label: action.payload.label  
 }  
 }  
 });  
}

### 3. Effect-Based Side Effects

**HTTP calls as declarative effects:** - Actions dispatch effects - Effects handle async operations - Success/error actions update state - Clean separation of concerns

**Flow:**

User Action → Dispatch Action → Dispatch Effect → HTTP Call  
 ↓ ↓  
 Update State Success/Error Action  
 ↓ ↓  
 Re-render Update State → Re-render

### 4. Change Tracking Pattern

**Local changes before save:** - User makes changes (stored in \*Changes state) - UI shows unsaved indicator - Save button persists to backend - Success handler reloads data

**Benefits:** - Reduces API calls - Allows batch operations - Provides clear undo mechanism - Improves user experience

**Implementation:**

// Track changes  
'UPDATE\_ANSWER\_LABEL': (coeffects) => {  
 updateState({  
 answerChanges: {  
 ...state.answerChanges,  
 [answerId]: {  
 ...state.answerChanges[answerId],  
 label: newLabel  
 }  
 }  
 });  
},  
  
// Save all changes  
'SAVE\_QUESTION\_IMMEDIATELY': (coeffects) => {  
 const changes = state.questionChanges[questionId];  
 dispatch('UPDATE\_QUESTION\_API', {changes});  
  
 // Clear changes on success  
 updateState({  
 questionChanges: omit(state.questionChanges, questionId)  
 });  
}

### 5. Immutable State Updates

**Never mutate state directly:**

// ❌ WRONG  
state.questionChanges[id] = newValue;  
  
// ✅ CORRECT  
updateState({  
 questionChanges: {  
 ...state.questionChanges,  
 [id]: newValue  
 }  
});

**Rationale**: Ensures predictable updates and enables change detection

### 6. Component Composition

**Single large component vs. many small:** - Current: Monolithic component (21,166 lines in index.js) - Renders entire application - All view logic in one file

**Rationale**: - ServiceNow UI Core pattern - Simplifies state management - Reduces complexity of prop passing - Centralized action handling

**Trade-offs**: - Large file size - Harder to navigate - More merge conflicts - But: simpler state management, no prop drilling

## Data Flow Architecture

### Request Flow

UI Event (e.g., click Save)  
 ↓  
Action Dispatched: 'SAVE\_QUESTION\_IMMEDIATELY'  
 ↓  
Action Handler: Validates data, prepares request  
 ↓  
Effect Dispatched: 'MAKE\_UPDATE\_QUESTION\_REQUEST'  
 ↓  
HTTP Effect: POST to ServiceNow REST API  
 ↓  
ServiceNow API: /api/x\_cadal\_careiq\_b\_0/careiq\_api/update-question  
 ↓  
Script Include: CareIQServices.builderUpdateQuestion()  
 ↓  
External API: POST to CareIQ Platform  
 ↓  
Response: Returns updated question data  
 ↓  
ServiceNow API: Transforms and returns response  
 ↓  
Success Action: 'UPDATE\_QUESTION\_SUCCESS'  
 ↓  
Action Handler: Updates state with new data  
 ↓  
UI Re-render: Component reflects changes  
 ↓  
System Message: "Question updated successfully"

### State Update Flow

Initial State  
 ↓  
User Interaction → Dispatch Action  
 ↓  
Action Handler Executes  
 ↓  
updateState() Called with New State  
 ↓  
State Merged (shallow merge)  
 ↓  
Component Re-renders  
 ↓  
Virtual DOM Diff  
 ↓  
Real DOM Updated  
 ↓  
UI Reflects New State

## Critical Design Decisions

### 1. POST for All ServiceNow Endpoints

**Decision**: Use POST method for all ServiceNow REST API endpoints, even when backend uses GET

**Rationale**: - ServiceNow POST handling more robust - Easier to send complex payloads - Consistent pattern across all endpoints - request.body.data pattern works reliably

**Implementation**:

// Client-side effect (always POST)  
export const MAKE\_FETCH\_SECTIONS\_REQUEST = createHttpEffect(  
 '/api/.../get-sections',  
 {method: 'POST', dataParam: 'requestBody'}  
);  
  
// Server-side API  
(function process(request, response) {  
 var requestData = request.body.data; // POST data  
 // Call backend with GET  
 var result = careiqServices.getSections(requestData.assessmentId);  
})(request, response);

### 2. Direct Fields, No Data Wrapper

**Decision**: Component sends direct fields in request body; ServiceNow wraps automatically

**Pattern**:

// Component sends  
const requestBody = JSON.stringify({  
 answerId: '123', // Direct fields  
 guidelineId: '456'  
});  
  
// ServiceNow receives  
var requestData = request.body.data; // Wrapped by ServiceNow  
// requestData = {answerId: '123', guidelineId: '456'}

**Rationale**: Simplifies component code while leveraging ServiceNow’s automatic wrapping

### 3. Two-Step Question Creation

**Decision**: Create questions and answers in separate API calls

**Pattern**:

// Step 1: Create question  
'ADD\_QUESTION\_TO\_SECTION': (coeffects) => {  
 dispatch('MAKE\_ADD\_QUESTION\_TO\_SECTION\_REQUEST', {  
 questionData: {...} // No answers  
 });  
},  
  
// Step 2: Add answers (in success handler)  
'ADD\_QUESTION\_TO\_SECTION\_SUCCESS': (coeffects) => {  
 const questionId = action.payload.questionId;  
 if (state.pendingQuestionAnswers) {  
 dispatch('MAKE\_ADD\_ANSWERS\_TO\_QUESTION\_REQUEST', {  
 questionId: questionId,  
 answers: state.pendingQuestionAnswers  
 });  
 }  
}

**Rationale**: Backend API design requires separate calls; ensures question ID exists before adding answers

### 4. State-Based Context for Typeaheads

**Decision**: Store context in state rather than relying on effect meta parameters

**Pattern**:

'SEARCH\_ANSWERS': (coeffects) => {  
 // Store context in state  
 const answerSearchContext = {  
 contentType: 'answer',  
 answerId: answerId,  
 searchText: searchText  
 };  
  
 updateState({  
 answerTypeaheadLoading: true,  
 currentAnswerSearchContext: answerSearchContext // Stored!  
 });  
  
 dispatch('MAKE\_GENERIC\_TYPEAHEAD\_REQUEST', {  
 requestBody: {...},  
 meta: {contentType: 'answer'} // Meta params unreliable  
 });  
},  
  
'GENERIC\_TYPEAHEAD\_SUCCESS': (coeffects) => {  
 // Use stored context, not meta  
 const context = state.currentAnswerSearchContext;  
 if (context && context.contentType === 'answer') {  
 updateState({  
 answerTypeaheadResults: action.payload.results  
 });  
 }  
}

**Rationale**: Meta parameters can be undefined or lost; state-based context is reliable

### 5. Question isUnsaved Flag + Change Tracking

**Decision**: Use both questionChanges state AND isUnsaved flag on question objects

**Pattern**:

// Track changes in state  
'UPDATE\_QUESTION\_LABEL': (coeffects) => {  
 updateState({  
 questionChanges: {  
 ...state.questionChanges,  
 [questionId]: {label: newLabel}  
 }  
 });  
},  
  
// Set isUnsaved flag on question object  
'MARK\_QUESTION\_UNSAVED': (coeffects) => {  
 const updatedQuestions = state.currentQuestions.questions.map(q =>  
 q.ids.id === questionId ? {...q, isUnsaved: true} : q  
 );  
 updateState({  
 currentQuestions: {  
 ...state.currentQuestions,  
 questions: updatedQuestions  
 }  
 });  
},  
  
// Clear BOTH on save  
'SAVE\_QUESTION\_SUCCESS': (coeffects) => {  
 // Clear change tracking  
 const updatedChanges = omit(state.questionChanges, questionId);  
  
 // Clear isUnsaved flag  
 const updatedQuestions = state.currentQuestions.questions.map(q =>  
 q.ids.id === questionId ? {...q, isUnsaved: false} : q  
 );  
  
 updateState({  
 questionChanges: updatedChanges,  
 currentQuestions: {  
 ...state.currentQuestions,  
 questions: updatedQuestions  
 }  
 });  
}

**Rationale**: UI renders Save/Cancel buttons based on isUnsaved flag; change tracking stores actual modifications. Both must be managed.

### 6. Post-Save Reload Pattern

**Decision**: Always reload assessment data after save operations

**Pattern**:

'SAVE\_SECTION\_SUCCESS': (coeffects) => {  
 // Clear change tracking  
 updateState({  
 sectionChanges: {},  
 questionChanges: {},  
 answerChanges: {}  
 });  
  
 // Reload entire assessment  
 if (state.currentAssessmentId) {  
 dispatch('FETCH\_ASSESSMENT\_DETAILS', {  
 assessmentId: state.currentAssessmentId  
 });  
 }  
}

**Rationale**: - Ensures data consistency - Clears all unsaved flags - Gets latest data from backend - Prevents stale state issues

### 7. UUID vs Temporary ID Handling

**Decision**: Use temporary IDs for newly created items until backend assigns real UUIDs

**Pattern**:

'ADD\_NEW\_SECTION': (coeffects) => {  
 const tempId = 'temp\_' + Date.now();  
 updateState({  
 sections: [  
 ...state.sections,  
 {id: tempId, name: 'New Section', isNew: true}  
 ]  
 });  
},  
  
'SAVE\_SECTIONS': (coeffects) => {  
 state.sections.forEach(section => {  
 if (section.id.startsWith('temp\_')) {  
 // Use ADD API (don't send temp ID)  
 dispatch('ADD\_SECTION\_API', {name: section.name});  
 } else {  
 // Use UPDATE API (send real UUID)  
 dispatch('UPDATE\_SECTION\_API', {id: section.id, name: section.name});  
 }  
 });  
}

**Rationale**: Allows optimistic UI updates while ensuring backend receives valid UUIDs

## Performance Considerations

### 1. Virtual DOM Rendering

**Snabbdom’s Virtual DOM**: - Efficient diffing algorithm - Only updates changed DOM nodes - Minimal reflows and repaints

**Optimization**: - Large lists use keys for efficient updates - Conditional rendering reduces unnecessary DOM

### 2. Lazy Loading

**PGI Hierarchy**: - Problems load initially (collapsed) - Goals load on-demand when problem expanded - Interventions load on-demand when goal expanded

**Rationale**: Reduces initial payload and API calls

### 3. Debounced Search

**Typeahead searches**: - 300-500ms debounce on input - Prevents excessive API calls - Improves performance

**Implementation**:

'TYPEAHEAD\_INPUT\_CHANGE': (coeffects) => {  
 clearTimeout(state.typeaheadTimeout);  
 const timeout = setTimeout(() => {  
 dispatch('MAKE\_TYPEAHEAD\_REQUEST', {query});  
 }, 300);  
 updateState({typeaheadTimeout: timeout});  
}

### 4. Per-Item Loading States

**Concurrent operations**: - Loading state per item ID - Allows multiple saves simultaneously - Provides specific feedback

**Implementation**:

// Set loading  
updateState({  
 savingQuestions: {  
 ...state.savingQuestions,  
 [questionId]: true  
 }  
});  
  
// Clear loading  
const updated = {...state.savingQuestions};  
delete updated[questionId];  
updateState({savingQuestions: updated});

## Security Architecture

### 1. No Direct External API Calls

**All communication routed through ServiceNow**: - Client → ServiceNow REST API - ServiceNow → CareIQ Platform - Response path reversed

**Benefits**: - Credentials never exposed to client - Request validation server-side - Audit logging in ServiceNow - Rate limiting possible

### 2. ServiceNow Security Restrictions

**Restricted operations in catch blocks**:

// ❌ NOT ALLOWED  
catch (e) {  
 var msg = e.message; // Security violation  
 var stack = e.stack; // Security violation  
}  
  
// ✅ ALLOWED  
catch (e) {  
 var errorMsg = 'Server error occurred';  
 try {  
 if (e && typeof e.toString === 'function') {  
 errorMsg = e.toString();  
 }  
 } catch (innerE) {  
 errorMsg = 'Unexpected error';  
 }  
}

**Rationale**: ServiceNow security model prevents direct property access on error objects

### 3. Request Data Stashing

**Pattern**:

(function process(request, response) {  
 var requestData = request.body.data; // Stash immediately  
  
 // Don't access request.body.data again  
 var result = processData(requestData);  
})(request, response);

**Rationale**: Prevents data consumption issues in ServiceNow

## Scalability Considerations

### Current Limitations

**Component Size**: - 21,166 lines in single file - Can be difficult to navigate - Potential merge conflict issues

**State Size**: - Large state object (8,500+ lines definition) - All data in memory - May impact performance with very large assessments

### Future Enhancements

**Potential Improvements**: 1. Code splitting by feature 2. Lazy loading of sections 3. Pagination for large question lists 4. Service workers for offline capability 5. State persistence across sessions

## Integration Points

### External Systems

**CareIQ Platform**: - RESTful API - Authentication via OAuth/token - JSON data format - HTTPS only

**ServiceNow Platform**: - Scripted REST APIs - Script Includes - Application tables (if any) - ServiceNow security model

## Deployment Architecture

### Build Process

1. Source files in /src/cadal-careiq-builder/
2. Babel transpilation
3. SCSS compilation
4. Bundling by ServiceNow build system
5. Deployment to ServiceNow instance

### Version Management

**package.json**:

{  
 "version": "0.1.092",  
 "scopeName": "x\_cadal\_careiq\_b\_0"  
}

**Versioning Pattern**: Increment last digit (0.1.092 → 0.1.093)

## Monitoring and Observability

### System Messages

**User-facing notifications**: - Success messages (green) - Warning messages (yellow) - Error messages (red) - Displayed in ticker at top of screen

### Logging

**Browser console**: - JavaScript errors - Network requests - State changes (if debugging enabled)

**ServiceNow logs**: - API call logs - Script Include execution - Error logs

## Summary

CareIQ Builder is a sophisticated assessment authoring tool built on ServiceNow UI Core framework. Key architectural highlights:

1. **Security-First**: All external calls routed through ServiceNow
2. **Declarative**: State-based architecture with effect-driven side effects
3. **Change Tracking**: Local modifications before backend persistence
4. **Component-Based**: Single large component with comprehensive state management
5. **Integration**: Seamless connection to CareIQ platform for clinical content

The architecture prioritizes security, maintainability, and user experience while working within ServiceNow platform constraints.

## Next Sections

This document continues with: - Component Structure (detailed breakdown) - State Management (comprehensive state documentation) - API Communication Patterns (request/response flows) - UI Patterns and Components (rendering strategies) - Critical Implementation Patterns (code examples) - Security and Error Handling (detailed patterns) - Testing and Deployment (procedures)