# Critical Implementation Patterns

## Overview

This document details the most critical implementation patterns in CareIQ Builder. These patterns are essential to understanding how the application works and must be followed when making changes.

**Reference**: Many patterns documented in CLAUDE.md at project root.

## 1. Two-Step Question Creation Pattern

### Problem

The CareIQ backend API requires questions and answers to be created in separate API calls: 1. Create question (returns question ID) 2. Add answers to question (requires question ID)

### Solution

**Step 1**: Create question without answers, store answers in state

'ADD\_QUESTION\_TO\_SECTION': (coeffects) => {  
 const {action, state, updateState, dispatch} = coeffects;  
 const {questionData, answers} = action.payload;  
  
 // Store answers for step 2  
 updateState({  
 pendingQuestionAnswers: answers  
 });  
  
 // Create question only (no answers)  
 const requestBody = JSON.stringify({  
 section\_id: state.selectedSection,  
 label: questionData.label,  
 type: questionData.type,  
 voice: questionData.voice,  
 gt\_id: state.currentAssessmentId  
 });  
  
 dispatch('ADD\_QUESTION\_TO\_SECTION\_API', {requestBody});  
}

**Step 2**: Add answers in success handler

'ADD\_QUESTION\_TO\_SECTION\_SUCCESS': (coeffects) => {  
 const {action, state, updateState, dispatch} = coeffects;  
  
 // Extract question ID from response  
 const questionId = action.payload.questionId;  
  
 // Add answers if any were pending  
 if (state.pendingQuestionAnswers && state.pendingQuestionAnswers.length > 0) {  
 const requestBody = JSON.stringify({  
 question\_id: questionId,  
 answers: state.pendingQuestionAnswers  
 });  
  
 dispatch('MAKE\_ADD\_ANSWERS\_TO\_QUESTION\_REQUEST', {requestBody});  
  
 // Clear pending answers  
 updateState({  
 pendingQuestionAnswers: null  
 });  
 }  
}

**API Endpoints**: - Step 1: POST /api/.../add-question-to-section → POST /builder/section/{id}/questions - Step 2: POST /api/.../add-answers-to-question → POST /builder/question/{id}/answers

**DEPRECATED** (Do Not Use): - ❌ ADD\_QUESTION\_API action (old single-step pattern) - ❌ /api/.../add-question endpoint (removed) - ❌ builderAddQuestion() method that sends question + answers in one call

**Why This Pattern**: - Backend expects separate API calls - Question must exist before answers can be added - Question ID needed to associate answers correctly

## 2. Save/Cancel Button Display Pattern

### Problem

Save/Cancel buttons must disappear after saving, but clearing only questionChanges state doesn’t hide them because buttons are rendered based on question.isUnsaved property.

### Root Cause

Buttons rendered based on question.isUnsaved flag in question object:

// View layer (around line ~1461)  
{question.isUnsaved && [  
 <button onclick={() => dispatch('SAVE\_QUESTION\_IMMEDIATELY', {questionId: question.ids.id})}>  
 💾 Save  
 </button>,  
 <button onclick={() => dispatch('CANCEL\_QUESTION\_CHANGES', {questionId: question.ids.id})}>  
 ↶ Cancel  
 </button>  
]}

### Solution

Clear BOTH change tracking AND the isUnsaved flag:

'SAVE\_QUESTION\_IMMEDIATELY': (coeffects) => {  
 const {action, state, updateState, dispatch} = coeffects;  
 const {questionId} = action.payload;  
  
 // ... validation and API call ...  
},  
  
'UPDATE\_QUESTION\_SUCCESS': (coeffects) => {  
 const {action, state, updateState, dispatch} = coeffects;  
 const questionId = action.payload.questionId;  
  
 // 1. Clear change tracking  
 const updatedQuestionChanges = {...state.questionChanges};  
 delete updatedQuestionChanges[questionId];  
  
 const updatedAnswerChanges = {...state.answerChanges};  
 // Remove all answer changes for this question's answers  
 // ... (implementation details)  
  
 // 2. CRITICAL: Clear isUnsaved flag on question object  
 const updatedQuestions = state.currentQuestions.questions.map(q =>  
 q.ids.id === questionId ? {...q, isUnsaved: false} : q  
 );  
  
 updateState({  
 questionChanges: updatedQuestionChanges,  
 answerChanges: updatedAnswerChanges,  
 currentQuestions: {  
 ...state.currentQuestions,  
 questions: updatedQuestions  
 }  
 });  
  
 // 3. Post-save reload pattern  
 if (state.currentAssessmentId) {  
 dispatch('FETCH\_ASSESSMENT\_DETAILS', {  
 assessmentId: state.currentAssessmentId  
 });  
 }  
}

**Key Insight**: Clearing questionChanges only affects change tracking state. The UI buttons check question.isUnsaved directly. Both must be cleared.

**Location in Code**: src/cadal-careiq-builder/index.js:~1461 (button rendering), action handlers around line 12,000+

## 3. State-Based Typeahead Context Pattern

### Problem

Effect meta parameters can be undefined or lost, causing typeahead results to display in wrong component or not at all.

### Solution

Store search context in state before dispatching effect:

'SEARCH\_ANSWERS': (coeffects) => {  
 const {action, state, updateState, dispatch} = coeffects;  
 const {answerId, searchText} = action.payload;  
  
 // 1. Create context object  
 const answerSearchContext = {  
 contentType: 'answer',  
 answerId: answerId,  
 searchText: searchText,  
 timestamp: Date.now()  
 };  
  
 // 2. Store context in state  
 updateState({  
 answerTypeaheadLoading: true,  
 currentAnswerSearchContext: answerSearchContext // CRITICAL  
 });  
  
 // 3. Dispatch effect (meta may not work)  
 dispatch('MAKE\_GENERIC\_TYPEAHEAD\_REQUEST', {  
 requestBody: JSON.stringify({  
 contentType: 'answer',  
 searchText: searchText  
 }),  
 meta: {contentType: 'answer'} // Don't rely on this  
 });  
}

Use stored context in success handler:

'GENERIC\_TYPEAHEAD\_SUCCESS': (coeffects) => {  
 const {action, state, updateState} = coeffects;  
 const results = action.payload.results || [];  
  
 // Use stored context, NOT meta params  
 const answerContext = state.currentAnswerSearchContext;  
 const questionContext = state.currentQuestionSearchContext;  
 const sectionContext = state.currentSectionSearchContext;  
  
 if (answerContext && answerContext.contentType === 'answer') {  
 updateState({  
 answerTypeaheadResults: results,  
 answerTypeaheadLoading: false,  
 answerTypeaheadVisible: true  
 });  
 // Don't clear context - let blur/escape events handle it  
 } else if (questionContext && questionContext.contentType === 'question') {  
 updateState({  
 questionTypeaheadResults: results,  
 questionTypeaheadLoading: false,  
 questionTypeaheadVisible: true  
 });  
 } else if (sectionContext && sectionContext.contentType === 'section') {  
 updateState({  
 sectionTypeaheadResults: results,  
 sectionTypeaheadLoading: false,  
 sectionTypeaheadVisible: true  
 });  
 }  
}

Clear context when typeahead closes:

'ANSWER\_TYPEAHEAD\_HIDE': (coeffects) => {  
 const {updateState} = coeffects;  
  
 updateState({  
 answerTypeaheadVisible: false,  
 answerTypeaheadResults: [],  
 currentAnswerSearchContext: null // Clear context  
 });  
}

**Used By**: All typeaheads (questions, answers, sections, goals, interventions)

**Why This Pattern Works**: - State-based context is reliable (always accessible) - Prevents stuck loading states - Enables proper result routing - Documented in CLAUDE.md as “working pattern”

## 4. Post-Save Reload Pattern

### Problem

After saving changes, local state may be out of sync with backend (IDs, flags, relationships).

### Solution

Always reload assessment data after successful save:

'[ANY\_SAVE\_OPERATION]\_SUCCESS': (coeffects) => {  
 const {state, updateState, dispatch} = coeffects;  
  
 // 1. Clear change tracking  
 updateState({  
 sectionChanges: {},  
 questionChanges: {},  
 answerChanges: {},  
 scoringChanges: {}  
 });  
  
 // 2. Show success message  
 dispatch('ADD\_SYSTEM\_MESSAGE', {  
 type: 'success',  
 message: 'Changes saved successfully!'  
 });  
  
 // 3. CRITICAL: Reload assessment data  
 if (state.currentAssessmentId) {  
 dispatch('FETCH\_ASSESSMENT\_DETAILS', {  
 assessmentId: state.currentAssessmentId  
 });  
 }  
}

**What Gets Reloaded**: - All sections - All questions for selected section - Backend-assigned IDs for new items - Cleared isUnsaved flags - Updated relationships

**Why This Pattern**: - Ensures consistency with backend - Gets new IDs for newly created items - Clears all unsaved indicators - Refreshes all data including relationships

**Operations Using This Pattern**: - Save question - Save answer - Save section - Delete question/answer/section - Add/delete relationships - Any operation modifying assessment data

## 5. Assessment ID Storage Pattern

### Problem

Need assessment ID for API calls, but accessing state.currentAssessment.id is fragile if assessment object structure changes.

### Solution

Store assessment ID separately:

'OPEN\_ASSESSMENT\_BUILDER': (coeffects) => {  
 const {action, updateState, dispatch} = coeffects;  
 const {assessment} = action.payload;  
  
 // Store BOTH assessment object AND ID separately  
 updateState({  
 builderView: true,  
 currentAssessment: assessment,  
 currentAssessmentId: assessment.id // Separate storage  
 });  
  
 // Load assessment details  
 dispatch('FETCH\_ASSESSMENT\_DETAILS', {  
 assessmentId: assessment.id  
 });  
}

Use stored ID for API calls:

'SAVE\_QUESTION': (coeffects) => {  
 const {state, dispatch} = coeffects;  
  
 const requestBody = JSON.stringify({  
 question\_id: questionId,  
 gt\_id: state.currentAssessmentId, // Use stored ID, not state.currentAssessment.id  
 // ...  
 });  
  
 dispatch('MAKE\_UPDATE\_QUESTION\_REQUEST', {requestBody});  
}

**Why This Pattern**: - Simpler access (state.currentAssessmentId vs state.currentAssessment?.id) - Avoids null/undefined errors - Decouples ID from assessment object structure - Documented in CLAUDE.md

## 6. UUID vs Temporary ID Handling

### Problem

Backend APIs expect real UUIDs, not temporary IDs assigned locally.

### Solution

Check ID format and route to appropriate API:

'SAVE\_SECTION': (coeffects) => {  
 const {action, state, dispatch} = coeffects;  
 const {sectionId} = action.payload;  
  
 const sectionData = state.sectionChanges[sectionId];  
  
 // Check if new (temp ID) or existing (UUID)  
 if (sectionData.action === 'add' || sectionId.startsWith('temp\_')) {  
 // New section - use ADD API (don't send temp ID)  
 const requestBody = JSON.stringify({  
 guideline\_template\_id: state.currentAssessmentId,  
 name: sectionData.name,  
 parent\_id: sectionData.parent\_id || null,  
 sort\_order: sectionData.sort\_order  
 });  
  
 dispatch('MAKE\_ADD\_SECTION\_REQUEST', {requestBody});  
 } else {  
 // Existing section - use UPDATE API (send real UUID)  
 const requestBody = JSON.stringify({  
 section\_id: sectionId, // Real UUID  
 name: sectionData.name,  
 parent\_id: sectionData.parent\_id || null,  
 sort\_order: sectionData.sort\_order  
 });  
  
 dispatch('MAKE\_SECTION\_UPDATE\_REQUEST', {requestBody, sectionId});  
 }  
}

**Temporary ID Pattern**: temp\_${Date.now()} or similar

**Why This Pattern**: - Allows optimistic UI updates - Backend assigns real IDs - Different APIs for add vs update

## 7. State-Based PGI Refresh Pattern

### Problem

Effect meta parameters don’t reliably pass context for refreshing PGI data after operations.

### Solution

Store context in state before operation:

'SAVE\_INTERVENTION\_TO\_GOAL': (coeffects) => {  
 const {action, state, updateState, dispatch} = coeffects;  
 const {goalId, interventionData} = action.payload;  
  
 // Store goal ID for refresh after success  
 updateState({  
 lastAddedInterventionGoalId: goalId,  
 savingInterventions: {  
 ...state.savingInterventions,  
 [goalId]: true  
 }  
 });  
  
 const requestBody = JSON.stringify({  
 goal\_id: goalId,  
 description: interventionData.description,  
 type: interventionData.type  
 });  
  
 dispatch('MAKE\_ADD\_INTERVENTION\_REQUEST', {requestBody});  
}

Use stored context in success handler:

'ADD\_INTERVENTION\_SUCCESS': (coeffects) => {  
 const {action, state, updateState, dispatch} = coeffects;  
  
 // Use stored goal ID  
 const goalId = state.lastAddedInterventionGoalId;  
  
 // Clear loading state  
 const updatedSaving = {...state.savingInterventions};  
 delete updatedSaving[goalId];  
  
 updateState({  
 savingInterventions: updatedSaving,  
 lastAddedInterventionGoalId: null // Clear after use  
 });  
  
 // Refresh interventions for this goal  
 if (goalId && state.currentAssessmentId) {  
 dispatch('LOAD\_GOAL\_INTERVENTIONS', {  
 goalId: goalId,  
 guidelineTemplateId: state.currentAssessmentId  
 });  
 }  
  
 // Show success message  
 dispatch('ADD\_SYSTEM\_MESSAGE', {  
 type: 'success',  
 message: 'Intervention added successfully'  
 });  
}

**Also Refresh Modal Relationships**:

'ADD\_GOAL\_SUCCESS': (coeffects) => {  
 const {state, dispatch} = coeffects;  
  
 // 1. Refresh modal relationships if open  
 if (state.relationshipModalAnswerId) {  
 dispatch('LOAD\_ANSWER\_RELATIONSHIPS', {  
 answerId: state.relationshipModalAnswerId  
 });  
 }  
  
 // 2. Refresh hierarchical data for expanded items  
 Object.keys(state.expandedProblems || {}).forEach(problemId => {  
 if (state.expandedProblems[problemId] === true) {  
 dispatch('LOAD\_PROBLEM\_GOALS', {  
 problemId: problemId,  
 guidelineTemplateId: state.currentAssessmentId  
 });  
 }  
 });  
}

**Why This Pattern**: - Meta parameters unreliable for PGI operations - State-based refresh ensures correct data reloaded - Documented in CLAUDE.md

## 8. Per-Item Loading State Pattern

### Problem

Multiple items can be saving simultaneously. Need specific loading indicator for each.

### Solution

Use object with item IDs as keys:

// Initial state  
{  
 savingQuestions: {}, // {questionId: true}  
 deletingAnswers: {}, // {answerId: true}  
 savingGoals: {} // {problemId: true}  
}  
  
// Set loading state  
'SAVE\_QUESTION': (coeffects) => {  
 const {action, state, updateState, dispatch} = coeffects;  
 const {questionId} = action.payload;  
  
 updateState({  
 savingQuestions: {  
 ...state.savingQuestions,  
 [questionId]: true // Set loading for this specific question  
 }  
 });  
  
 dispatch('MAKE\_UPDATE\_QUESTION\_REQUEST', {...});  
}  
  
// Clear loading state  
'UPDATE\_QUESTION\_SUCCESS': (coeffects) => {  
 const {action, state, updateState} = coeffects;  
 const questionId = action.payload.questionId;  
  
 // Remove loading state for this question  
 const updatedSavingQuestions = {...state.savingQuestions};  
 delete updatedSavingQuestions[questionId];  
  
 updateState({  
 savingQuestions: updatedSavingQuestions  
 });  
}  
  
// Render loading overlay  
{state.savingQuestions[question.ids.id] && (  
 <LoadingOverlay message="Saving question..." />  
)}

**Benefits**: - Multiple items can save concurrently - Specific feedback per item - Doesn’t block other operations - User knows exactly which item is saving

## 9. System Message Pattern

### Problem

Need to surface backend messages and errors to users.

### Solution

Always add system messages for operations:

'OPERATION\_SUCCESS': (coeffects) => {  
 const {action, state, updateState, dispatch} = coeffects;  
  
 // Extract message from backend response  
 let systemMessage = 'Operation completed successfully!';  
 let messageType = 'success';  
  
 if (action.payload && action.payload.detail) {  
 systemMessage = action.payload.detail;  
  
 // Detect warnings in message  
 if (systemMessage.toLowerCase().includes('duplicate')) {  
 messageType = 'warning';  
 } else if (systemMessage.toLowerCase().includes('error')) {  
 messageType = 'error';  
 }  
 }  
  
 // Add to system messages  
 updateState({  
 systemMessages: [  
 ...(state.systemMessages || []),  
 {  
 id: 'msg-' + Date.now(),  
 type: messageType,  
 message: systemMessage,  
 timestamp: new Date().toISOString()  
 }  
 ]  
 });  
  
 // Auto-dismiss after 5 seconds (optional)  
 setTimeout(() => {  
 dispatch('DISMISS\_SYSTEM\_MESSAGE', {id: 'msg-' + Date.now()});  
 }, 5000);  
}

**Message Types**: - success - Green - error - Red - warning - Yellow - info - Blue

**Why This Pattern**: - Backend can return useful messages (duplicate checks, validation errors) - User gets immediate feedback - Messages can be warnings even on successful API calls - Centralized message display

## 10. ServiceNow API Data Pattern

### Problem

ServiceNow automatically wraps POST data in .data property, which can cause confusion.

### Solution

**Client sends direct fields (no wrapper)**:

// Client code  
const requestBody = JSON.stringify({  
 answerId: '123', // Direct fields  
 guidelineId: '456'  
});  
  
dispatch('MAKE\_API\_REQUEST', {requestBody});

**ServiceNow receives wrapped data**:

// ServiceNow API  
(function process(request, response) {  
 // CRITICAL: Stash immediately  
 var requestData = request.body.data; // Data is wrapped by ServiceNow  
  
 var answerId = requestData.answerId; // Access from wrapped data  
 var guidelineId = requestData.guidelineId;  
  
 // ...  
})(request, response);

**Why This Pattern**: - ServiceNow does wrapping automatically - Component code simpler (no manual wrapping) - Consistent with ServiceNow conventions - Documented in CLAUDE.md

## 11. ServiceNow Safe Error Handling

### Problem

Cannot access e.message or e.stack in ServiceNow catch blocks (security restriction).

### Solution

Use safe error handling pattern:

(function process(request, response) {  
 var requestData = request.body.data;  
  
 try {  
 var careiqServices = new x\_1628056\_careiq.CareIQServices();  
 var result = careiqServices.someMethod(requestData);  
  
 response.setStatus(200);  
 response.setBody(result);  
  
 } catch (e) {  
 // SAFE ERROR HANDLING (no direct property access)  
 var errorMsg = 'Unexpected server error occurred';  
  
 try {  
 // Safe: call toString method if available  
 if (e && typeof e.toString === 'function') {  
 errorMsg = e.toString();  
 }  
 } catch (innerE) {  
 // Even toString failed, use generic message  
 errorMsg = 'Server error occurred';  
 }  
  
 response.setStatus(500);  
 response.setBody({  
 success: false,  
 message: errorMsg  
 });  
 }  
})(request, response);

**Why This Pattern**: - ServiceNow security model prevents direct error property access - e.toString() is allowed - Provides meaningful error messages when possible - Graceful fallback for worst cases

## 12. Script Include Instantiation Pattern

### Problem

Script Includes must be instantiated correctly with proper scope.

### Solution

Use correct scope prefix:

// ✅ CORRECT  
var careiqServices = new x\_1628056\_careiq.CareIQServices();  
  
// ❌ WRONG - Missing scope  
var careiqServices = new CareIQServices();  
  
// ❌ WRONG - Wrong scope  
var careiqServices = new x\_cadal\_careiq\_b\_0.CareIQServices();

**Scope**: All CareIQ Script Includes are in x\_1628056\_careiq scope (CareIQ services scope, not builder scope)

**Builder Scope**: x\_cadal\_careiq\_b\_0 (for builder-specific code)

**Why This Pattern**: - Services are in separate application scope - Enables code reuse across applications - Proper encapsulation

## 13. Select Element Pattern (ServiceNow UI Core)

### Problem

ServiceNow’s snabbdom renderer doesn’t support value attribute on <select> elements.

### Solution

Use selected attribute on <option> elements:

// ❌ WRONG - value on select  
<select value={question.type}>  
 <option value="Single Select">Single Select</option>  
 <option value="Multiselect">Multiselect</option>  
</select>  
  
// ✅ CORRECT - selected on options  
<select onchange={(e) => dispatch('UPDATE\_TYPE', {type: e.target.value})}>  
 <option value="Single Select" selected={question.type === 'Single Select'}>  
 Single Select  
 </option>  
 <option value="Multiselect" selected={question.type === 'Multiselect'}>  
 Multiselect  
 </option>  
 <option value="Free Text" selected={question.type === 'Free Text'}>  
 Free Text  
 </option>  
</select>

**Why This Pattern**: - ServiceNow UI Core limitation - Works with snabbdom renderer - Documented in CLAUDE.md

## 14. Typeahead Dropdown Close Pattern

### Problem

Typeahead dropdown needs to close on click outside and Escape key, but allow item selection.

### Solution

Use blur delay and escape handling:

<input  
 type="text"  
 value={state.typeaheadQuery}  
 oninput={(e) => dispatch('TYPEAHEAD\_INPUT', {text: e.target.value})}  
 onkeydown={(e) => {  
 if (e.key === 'Escape') {  
 // Close immediately on Escape  
 dispatch('TYPEAHEAD\_HIDE');  
 } else if (e.key === 'Enter') {  
 // Select first result on Enter  
 if (state.typeaheadResults.length > 0) {  
 dispatch('TYPEAHEAD\_SELECT', {item: state.typeaheadResults[0]});  
 }  
 }  
 }}  
 onblur={(e) => {  
 // Delay close to allow click on dropdown item  
 setTimeout(() => {  
 dispatch('TYPEAHEAD\_HIDE');  
 }, 150);  
 }}  
/>

**Hide Action** (clears results and context):

'TYPEAHEAD\_HIDE': (coeffects) => {  
 const {updateState} = coeffects;  
  
 updateState({  
 typeaheadVisible: false,  
 typeaheadResults: [],  
 currentTypeaheadContext: null // Clear context  
 });  
}

**Why 150ms Delay**: - Allows onclick event on dropdown item to fire - Long enough for click, short enough to feel responsive - Prevents dropdown closing before selection

## Summary of Critical Patterns

1. **Two-Step Question Creation**: Separate API calls for question and answers
2. **Save Button Display**: Clear both questionChanges and isUnsaved flag
3. **Typeahead Context**: Store in state, not meta params
4. **Post-Save Reload**: Always reload data after saves
5. **Assessment ID Storage**: Store separately from assessment object
6. **UUID vs Temp ID**: Route to add/update APIs accordingly
7. **PGI Refresh**: State-based context for reliable refresh
8. **Per-Item Loading**: Object with item IDs as keys
9. **System Messages**: Surface backend messages to users
10. **ServiceNow Data Pattern**: Direct fields in, wrapped on server
11. **Safe Error Handling**: Use toString(), not message/stack
12. **Script Include Scope**: x\_1628056\_careiq.CareIQServices
13. **Select Elements**: Use selected on options, not value on select
14. **Typeahead Close**: Blur delay + escape key

These patterns are essential to CareIQ Builder’s correct operation. Deviating from them will cause bugs.