# Testing and Deployment

## Overview

This document covers testing strategies, quality assurance processes, and deployment procedures for CareIQ Builder.

## Testing Strategy

### Testing Pyramid

┌──────────────┐  
 │ Manual/E2E │ <- Fewer, high-value  
 │ Testing │  
 └──────────────┘  
 ┌────────────────────┐  
 │ Integration Tests │ <- Service + API tests  
 └────────────────────┘  
 ┌──────────────────────────┐  
 │ Unit Tests │ <- Many, fast tests  
 └──────────────────────────┘

**Recommended Distribution**: - 70% Unit Tests (action handlers, utilities) - 20% Integration Tests (API interactions, data flow) - 10% End-to-End Tests (critical user workflows)

## Unit Testing

### 1. Testing Action Handlers

**Setup**:

// Using Jest or similar framework  
import {actionHandlers} from './index.js';  
  
describe('SAVE\_QUESTION\_IMMEDIATELY', () => {  
 it('should dispatch update request with correct data', () => {  
 // Arrange  
 const mockCoeffects = {  
 action: {  
 type: 'SAVE\_QUESTION\_IMMEDIATELY',  
 payload: {questionId: 'question-123'}  
 },  
 state: {  
 questionChanges: {  
 'question-123': {  
 label: 'Updated question',  
 type: 'Single Select',  
 voice: 'Patient'  
 }  
 },  
 currentAssessmentId: 'assessment-456',  
 selectedSection: 'section-789'  
 },  
 updateState: jest.fn(),  
 dispatch: jest.fn()  
 };  
  
 // Act  
 actionHandlers['SAVE\_QUESTION\_IMMEDIATELY'](mockCoeffects);  
  
 // Assert  
 expect(mockCoeffects.updateState).toHaveBeenCalledWith({  
 savingQuestions: {'question-123': true}  
 });  
  
 expect(mockCoeffects.dispatch).toHaveBeenCalledWith(  
 'MAKE\_UPDATE\_QUESTION\_REQUEST',  
 expect.objectContaining({  
 requestBody: expect.any(String)  
 })  
 );  
  
 // Verify request body content  
 const dispatchCall = mockCoeffects.dispatch.mock.calls[0];  
 const requestBody = JSON.parse(dispatchCall[1].requestBody);  
 expect(requestBody.question\_id).toBe('question-123');  
 expect(requestBody.label).toBe('Updated question');  
 });  
  
 it('should not dispatch if no changes', () => {  
 const mockCoeffects = {  
 action: {  
 type: 'SAVE\_QUESTION\_IMMEDIATELY',  
 payload: {questionId: 'question-123'}  
 },  
 state: {  
 questionChanges: {}, // No changes  
 currentAssessmentId: 'assessment-456'  
 },  
 updateState: jest.fn(),  
 dispatch: jest.fn()  
 };  
  
 actionHandlers['SAVE\_QUESTION\_IMMEDIATELY'](mockCoeffects);  
  
 // Should show warning, not dispatch API request  
 expect(mockCoeffects.dispatch).toHaveBeenCalledWith(  
 'ADD\_SYSTEM\_MESSAGE',  
 expect.objectContaining({  
 type: 'warning'  
 })  
 );  
 expect(mockCoeffects.dispatch).not.toHaveBeenCalledWith(  
 'MAKE\_UPDATE\_QUESTION\_REQUEST',  
 expect.anything()  
 );  
 });  
});

### 2. Testing Utility Functions

**Example**:

import {calculateVisibleQuestions} from './utils.js';  
  
describe('calculateVisibleQuestions', () => {  
 it('should return all questions when no selections', () => {  
 const questions = [  
 {ids: {id: 'q1'}, triggeredBy: []},  
 {ids: {id: 'q2'}, triggeredBy: []}  
 ];  
 const selectedAnswers = {};  
 const relationshipMap = {};  
  
 const result = calculateVisibleQuestions(questions, selectedAnswers, relationshipMap);  
  
 expect(result).toHaveLength(2);  
 expect(result.map(q => q.ids.id)).toEqual(['q1', 'q2']);  
 });  
  
 it('should show triggered question when answer selected', () => {  
 const questions = [  
 {ids: {id: 'q1'}, triggeredBy: []},  
 {ids: {id: 'q2'}, triggeredBy: ['answer-1'], sourceQuestionId: 'q1'}  
 ];  
 const selectedAnswers = {  
 'q1': ['answer-1'] // Answer that triggers q2  
 };  
 const relationshipMap = {};  
  
 const result = calculateVisibleQuestions(questions, selectedAnswers, relationshipMap);  
  
 expect(result).toHaveLength(2);  
 expect(result.map(q => q.ids.id)).toEqual(['q1', 'q2']);  
 });  
  
 it('should hide triggered question when answer not selected', () => {  
 const questions = [  
 {ids: {id: 'q1'}, triggeredBy: []},  
 {ids: {id: 'q2'}, triggeredBy: ['answer-1'], sourceQuestionId: 'q1'}  
 ];  
 const selectedAnswers = {  
 'q1': ['answer-2'] // Different answer, doesn't trigger q2  
 };  
 const relationshipMap = {};  
  
 const result = calculateVisibleQuestions(questions, selectedAnswers, relationshipMap);  
  
 expect(result).toHaveLength(1);  
 expect(result[0].ids.id).toBe('q1');  
 });  
});

### 3. Testing Components

**Testing Functional Components**:

import {CheckIcon, SpinnerIcon} from './index.js';  
import {render} from '@testing-library/snabbdom'; // Hypothetical  
  
describe('CheckIcon', () => {  
 it('should render SVG with correct attributes', () => {  
 const vnode = CheckIcon();  
  
 expect(vnode.sel).toBe('svg');  
 expect(vnode.data.attrs.width).toBe('14');  
 expect(vnode.data.attrs.height).toBe('14');  
 });  
});  
  
describe('SpinnerIcon', () => {  
 it('should use default size when not specified', () => {  
 const vnode = SpinnerIcon({});  
  
 expect(vnode.data.attrs.width).toBe('24');  
 expect(vnode.data.attrs.height).toBe('24');  
 });  
  
 it('should use custom size when specified', () => {  
 const vnode = SpinnerIcon({size: '48'});  
  
 expect(vnode.data.attrs.width).toBe('48');  
 expect(vnode.data.attrs.height).toBe('48');  
 });  
});

## Integration Testing

### 1. API Integration Tests

**Testing Complete Request/Response Flow**:

describe('Question Save Integration', () => {  
 let mockApiServer;  
  
 beforeEach(() => {  
 // Setup mock API server  
 mockApiServer = setupMockServer();  
 });  
  
 afterEach(() => {  
 mockApiServer.close();  
 });  
  
 it('should save question and reload data', async () => {  
 // Mock API responses  
 mockApiServer.mock('/api/.../update-question', {  
 status: 200,  
 body: {  
 success: true,  
 message: 'Question updated successfully',  
 data: {questionId: 'question-123'}  
 }  
 });  
  
 mockApiServer.mock('/api/.../get-sections', {  
 status: 200,  
 body: {  
 success: true,  
 data: {sections: [...]}  
 }  
 });  
  
 // Dispatch save action  
 dispatch('SAVE\_QUESTION\_IMMEDIATELY', {questionId: 'question-123'});  
  
 // Wait for API call  
 await waitFor(() => {  
 expect(mockApiServer.requests).toContainEqual(  
 expect.objectContaining({  
 url: '/api/.../update-question',  
 method: 'POST'  
 })  
 );  
 });  
  
 // Verify success handler executed  
 await waitFor(() => {  
 expect(state.questionChanges['question-123']).toBeUndefined();  
 });  
  
 // Verify reload was triggered  
 expect(mockApiServer.requests).toContainEqual(  
 expect.objectContaining({  
 url: '/api/.../get-sections',  
 method: 'POST'  
 })  
 );  
 });  
  
 it('should handle API error gracefully', async () => {  
 // Mock error response  
 mockApiServer.mock('/api/.../update-question', {  
 status: 500,  
 body: {  
 success: false,  
 message: 'Server error'  
 }  
 });  
  
 dispatch('SAVE\_QUESTION\_IMMEDIATELY', {questionId: 'question-123'});  
  
 await waitFor(() => {  
 expect(state.systemMessages).toContainEqual(  
 expect.objectContaining({  
 type: 'error',  
 message: expect.stringContaining('Server error')  
 })  
 );  
 });  
  
 // Verify loading state cleared  
 expect(state.savingQuestions['question-123']).toBeUndefined();  
 });  
});

### 2. ServiceNow API Tests

**Testing Server-Side Logic**:

describe('ServiceNow REST API: update-question', () => {  
 it('should validate required fields', () => {  
 const request = {  
 body: {  
 data: {} // Missing question\_id  
 }  
 };  
 const response = mockResponse();  
  
 process(request, response);  
  
 expect(response.status).toBe(400);  
 expect(response.body.success).toBe(false);  
 expect(response.body.message).toContain('question\_id');  
 });  
  
 it('should call CareIQ services with correct data', () => {  
 const mockServices = jest.fn().mockReturnValue({  
 success: true,  
 data: {}  
 });  
  
 const request = {  
 body: {  
 data: {  
 question\_id: 'question-123',  
 label: 'Updated question'  
 }  
 }  
 };  
 const response = mockResponse();  
  
 // Inject mock  
 global.x\_1628056\_careiq = {  
 CareIQServices: function() {  
 this.builderUpdateQuestion = mockServices;  
 }  
 };  
  
 process(request, response);  
  
 expect(mockServices).toHaveBeenCalledWith(  
 expect.objectContaining({  
 question\_id: 'question-123',  
 label: 'Updated question'  
 })  
 );  
 expect(response.status).toBe(200);  
 });  
});

## End-to-End Testing

### 1. Critical User Workflows

**Test Scenarios**:

1. **Create and Save Question**
   * Open assessment
   * Navigate to section
   * Add new question
   * Set question properties
   * Add answers
   * Save question
   * Verify question appears in list
2. **Edit Question with Triggered Question**
   * Select question
   * Add answer
   * Add triggered question relationship
   * Switch to Preview mode
   * Select answer
   * Verify triggered question appears
3. **Publish Assessment**
   * Complete assessment in Draft
   * Click Publish
   * Confirm publication
   * Verify status changes to Published
   * Verify cannot edit
4. **Create New Version**
   * Select published assessment
   * Click Create New Version
   * Verify new draft created
   * Make changes
   * Publish new version

### 2. E2E Test Example (Playwright/Cypress)

describe('Question Management E2E', () => {  
 beforeEach(() => {  
 cy.login();  
 cy.visit('/careiq-builder');  
 });  
  
 it('should create and save a new question', () => {  
 // Open assessment  
 cy.contains('Test Assessment').click();  
 cy.contains('Open').click();  
  
 // Select section  
 cy.get('.sections-panel').contains('Demographics').click();  
  
 // Add question  
 cy.contains('+ Add Question').click();  
  
 // Fill question details  
 cy.get('input[placeholder="Enter question text"]')  
 .type('What is your age?');  
  
 cy.get('select').select('Single Select');  
  
 // Add answers  
 cy.contains('+ Add Answer').click();  
 cy.get('.answer-item').first()  
 .find('input')  
 .type('18-25');  
  
 cy.contains('+ Add Answer').click();  
 cy.get('.answer-item').last()  
 .find('input')  
 .type('26-35');  
  
 // Save  
 cy.contains('💾 Save').click();  
  
 // Verify success  
 cy.contains('Question saved successfully', {timeout: 10000})  
 .should('be.visible');  
  
 // Verify question appears  
 cy.contains('What is your age?').should('be.visible');  
 cy.contains('18-25').should('be.visible');  
 });  
  
 it('should test conditional logic in preview mode', () => {  
 // Setup existing question with trigger  
 cy.setupQuestionWithTrigger();  
  
 // Switch to preview mode  
 cy.contains('Preview Mode').click();  
  
 // Initially, triggered question should not be visible  
 cy.contains('Triggered Question').should('not.exist');  
  
 // Select triggering answer  
 cy.contains('Yes').click();  
  
 // Triggered question should appear  
 cy.contains('Triggered Question').should('be.visible');  
  
 // Deselect triggering answer  
 cy.contains('No').click();  
  
 // Triggered question should disappear  
 cy.contains('Triggered Question').should('not.exist');  
 });  
});

## Manual Testing

### 1. Test Plan Template

**Test Case ID**: TC-001 **Title**: Create New Assessment **Priority**: High **Prerequisites**: User logged in, has create permissions

**Steps**: 1. Click “+ New Assessment” button 2. Enter assessment name: “Test Assessment” 3. Select use case category: “Chronic Care” 4. Select type: “Assessment Only” 5. Enter policy number: “TEST-001” 6. Click “Create”

**Expected Results**: - Success message displays - Assessment appears in list with “Draft” status - Assessment can be opened

**Actual Results**: [Filled during testing] **Status**: [Pass/Fail] **Notes**: [Any observations]

### 2. Regression Test Checklist

Before each release, test:

**Assessment Management**: - [ ] Create new assessment - [ ] Open existing assessment - [ ] Search assessments by name - [ ] Search assessments by policy number - [ ] View version history - [ ] Publish assessment - [ ] Create new version

**Section Management**: - [ ] Add section - [ ] Edit section name - [ ] Delete section (without children) - [ ] Add child section - [ ] Reorder sections - [ ] Section selection

**Question Management**: - [ ] Add question to section - [ ] Edit question label - [ ] Change question type - [ ] Change question voice - [ ] Delete question - [ ] Move question to different section - [ ] Question typeahead search

**Answer Management**: - [ ] Add answer to question - [ ] Edit answer label - [ ] Set secondary input type - [ ] Mark as mutually exclusive - [ ] Delete answer - [ ] Answer typeahead search

**Relationship Management**: - [ ] Add triggered question - [ ] Delete triggered question - [ ] Add guideline relationship - [ ] Delete guideline relationship - [ ] Add problem relationship - [ ] Expand problem to show goals - [ ] Add goal to problem - [ ] Expand goal to show interventions - [ ] Add intervention to goal

**Preview Mode**: - [ ] Switch to preview mode - [ ] Select answer (single select) - [ ] Select answers (multiselect) - [ ] Triggered question appears - [ ] Triggered question disappears - [ ] Switch back to edit mode

**UI/UX**: - [ ] System messages display - [ ] Loading indicators appear/disappear - [ ] Save/Cancel buttons appear when editing - [ ] Save/Cancel buttons disappear after save - [ ] Modal opens and closes - [ ] Dropdown opens and closes - [ ] Responsive design on mobile - [ ] No console errors

## Performance Testing

### 1. Load Testing

**Test Scenarios**: - Load assessment with 50+ sections - Load section with 100+ questions - Typeahead search with 1000+ results - Expand problem with 50+ goals

**Metrics to Measure**: - Initial page load time (< 3 seconds) - Assessment open time (< 2 seconds) - Section switch time (< 1 second) - API response time (< 1 second) - Typeahead response time (< 500ms)

### 2. Performance Optimization Checklist

* ☐ Virtual DOM efficiently updates
* ☐ Large lists use keys
* ☐ Conditional rendering prevents unnecessary renders
* ☐ Debouncing on typeahead searches
* ☐ Per-item loading states (not global blocks)
* ☐ Lazy loading for PGI hierarchy
* ☐ Minimal re-renders on state updates

## Deployment

### 1. Version Management

**Version Pattern**: 0.1.XXX

**Incrementing Version**:

// package.json  
{  
 "version": "0.1.092" // Current  
}  
  
// After changes  
{  
 "version": "0.1.093" // Incremented  
}

**When to Increment**: - After completing changes - Before deployment - Document changes in CLAUDE.md or changelog

### 2. Build Process

**Steps**: 1. Update version in package.json 2. Run build command (ServiceNow build system) 3. Babel transpilation 4. SCSS compilation 5. Bundling 6. Generate component artifacts

**Build Command** (if applicable):

npm run build

### 3. Deployment to ServiceNow

**ServiceNow Deployment Methods**:

**Option 1: Update Set** 1. Make changes in development instance 2. Create Update Set 3. Mark Update Set complete 4. Export Update Set 5. Import to target instance 6. Preview and commit

**Option 2: Source Control** 1. Commit changes to git repository 2. Push to remote 3. ServiceNow instance pulls from repository 4. Deploy via ServiceNow CI/CD

**Option 3: Manual Upload** 1. Build component locally 2. Upload files to ServiceNow 3. Update component record 4. Clear cache

### 4. Pre-Deployment Checklist

**Code Quality**: - [ ] All tests pass - [ ] No console errors - [ ] No linting errors - [ ] Code reviewed - [ ] Documentation updated

**Functionality**: - [ ] Regression tests pass - [ ] New features tested - [ ] Bug fixes verified - [ ] Performance acceptable

**Configuration**: - [ ] Version incremented - [ ] CLAUDE.md updated - [ ] Change log updated - [ ] Dependencies up to date

**Security**: - [ ] No hardcoded credentials - [ ] No sensitive data in logs - [ ] Error messages sanitized - [ ] Input validation in place

### 5. Deployment Steps

**Development → Test**: 1. Complete development work 2. Increment version 3. Run all tests 4. Create Update Set 5. Deploy to test instance 6. Smoke test critical workflows 7. Full regression test 8. Fix any issues found

**Test → Production**: 1. Get approval from stakeholders 2. Schedule deployment window 3. Notify users of maintenance 4. Deploy Update Set to production 5. Smoke test in production 6. Monitor for errors 7. Notify users deployment complete

### 6. Post-Deployment

**Immediate Checks**: - [ ] Application loads - [ ] Can create assessment - [ ] Can save question - [ ] No JavaScript errors - [ ] API calls successful

**Monitor for 24 Hours**: - Error logs - User reports - Performance metrics - System messages

**Rollback Plan**: - Keep previous Update Set - Document rollback procedure - Test rollback in non-production first

### 7. Rollback Procedure

**If Critical Issue Found**: 1. Notify stakeholders 2. Create rollback Update Set (or use previous version) 3. Deploy rollback 4. Verify system stable 5. Investigate root cause 6. Fix issue in development 7. Re-test 8. Re-deploy

## Continuous Integration/Continuous Deployment (CI/CD)

### Recommended CI/CD Pipeline

┌──────────────┐  
│ Git Commit │  
└──────┬───────┘  
 ↓  
┌──────────────┐  
│ Run Tests │ <- Unit, Integration  
└──────┬───────┘  
 ↓  
┌──────────────┐  
│ Build │ <- Transpile, Bundle  
└──────┬───────┘  
 ↓  
┌──────────────┐  
│ Deploy Dev │ <- Auto-deploy to dev instance  
└──────┬───────┘  
 ↓  
┌──────────────┐  
│ Smoke Test │ <- Basic health checks  
└──────┬───────┘  
 ↓ (manual trigger)  
┌──────────────┐  
│ Deploy Test │ <- Deploy to test instance  
└──────┬───────┘  
 ↓ (manual trigger + approval)  
┌──────────────┐  
│ Deploy Prod │ <- Deploy to production  
└──────────────┘

## Quality Gates

### Definition of Done

A feature is “done” when: - [ ] Code complete and reviewed - [ ] Unit tests written and passing - [ ] Integration tests passing - [ ] Manual testing complete - [ ] Documentation updated - [ ] No blocking bugs - [ ] Performance acceptable - [ ] Security review passed - [ ] Deployed to test environment - [ ] Stakeholder approval received

## Best Practices

### Testing

**DO**: ✅ Write tests before/during development ✅ Test happy path and error cases ✅ Use descriptive test names ✅ Mock external dependencies ✅ Test critical workflows end-to-end ✅ Run tests before committing ✅ Maintain test coverage above 70%

**DON’T**: ❌ Skip testing “simple” code ❌ Test implementation details ❌ Have flaky tests ❌ Rely only on manual testing ❌ Leave broken tests in codebase

### Deployment

**DO**: ✅ Increment version before deployment ✅ Document changes ✅ Test in non-production first ✅ Have rollback plan ✅ Schedule deployments during low-usage times ✅ Notify users of maintenance ✅ Monitor after deployment

**DON’T**: ❌ Deploy on Fridays or before holidays ❌ Skip testing in test environment ❌ Deploy without stakeholder approval ❌ Deploy multiple changes at once (hard to isolate issues) ❌ Forget to update documentation

## Summary

CareIQ Builder testing and deployment: - **Unit Tests**: Action handlers, utilities (70%) - **Integration Tests**: API flows (20%) - **E2E Tests**: Critical workflows (10%) - **Manual Testing**: Regression checklist - **Version Management**: Increment before deployment - **Deployment**: Dev → Test → Production - **Quality Gates**: Definition of done - **Monitoring**: Post-deployment checks - **Rollback**: Prepared and tested

This approach ensures quality, reliability, and smooth deployments.