

```
// Round to 2 decimal places
    return Math.round(ldl * 100) / 100;
} catch (error) {
    console.error('Error calculating LDL:', error);
    return null;
}
}

/***
 * Calculate non-HDL cholesterol
 * @param {number} tc - Total cholesterol
 * @param {number} hdl - HDL cholesterol
 * @returns {number|null} - Non-HDL or null if invalid inputs
 */
static calculateNonHDL(tc, hdl) {
    if (!tc || !hdl) {
        return null;
    }

    try {
        // Non-HDL = TC - HDL
        const nonHdl = tc - hdl;

        // Round to 2 decimal places
        return Math.round(nonHdl * 100) / 100;
    } catch (error) {
        console.error('Error calculating non-HDL:', error);
        return null;
    }
}

/***
 * Calculate BP standard deviation
 * @param {Array<number>} readings - BP readings
 * @returns {number|null} - Standard deviation or null if fewer than 2 readings
 */
static calculateBPStandardDeviation(readings) {
    if (!readings || readings.length < 2) {
        return null;
    }

    try {
```

```

// Calculate mean
const mean = readings.reduce((sum, val) => sum + val, 0) / readings.length;

// Calculate variance
const variance = readings.reduce((sum, val) => sum + Math.pow(val - mean, 2), 0) /
readings.length;

// Standard deviation is square root of variance
return Math.round(Math.sqrt(variance) * 10) / 10;
} catch (error) {
  console.error('Error calculating BP standard deviation:', error);
  return null;
}
}

/***
 * Convert between different units
 * @param {number} value - Value to convert
 * @param {string} fromUnit - From unit
 * @param {string} toUnit - To unit
 * @param {string} valueType - Type of value (e.g. 'cholesterol', 'height')
 * @returns {number} - Converted value
 */
static convertUnits(value, fromUnit, toUnit, valueType) {
  if (fromUnit === toUnit) {
    return value;
  }

  // Conversion factors for common unit types
  const factors = {
    // Cholesterol conversion (mmol/L <-> mg/dL)
    cholesterol: {
      'mmol/L_to_mg/dL': 38.67,
      'mg/dL_to_mmol/L': 0.02586
    },
    // HDL conversion (mmol/L <-> mg/dL)
    hdl: {
      'mmol/L_to_mg/dL': 38.67,
      'mg/dL_to_mmol/L': 0.02586
    },
    // LDL conversion (mmol/L <-> mg/dL)
    ldl: {
      'mmol/L_to_mg/dL': 38.67,
      'mg/dL_to_mmol/L': 0.02586
    }
  };
}

```

```

},
// Triglycerides conversion (mmol/L <-> mg/dL)
triglycerides: {
  'mmol/L_to_mg/dL': 88.5,
  'mg/dL_to_mmol/L': 0.01129
},
// Height conversion (cm <-> in)
height: {
  'cm_to_in': 0.393701,
  'in_to_cm': 2.54
},
// Weight conversion (kg <-> lb)
weight: {
  'kg_to_lb': 2.20462,
  'lb_to_kg': 0.453592
},
// Apolipoprotein B conversion (g/L <-> mg/dL)
apoB: {
  'g/L_to_mg/dL': 100,
  'mg/dL_to_g/L': 0.01
},
// Lipoprotein A conversion (mg/dL <-> nmol/L)
lipoproteinA: {
  'mg/dL_to_nmol/L': 2.5, // Approximate conversion
  'nmol/L_to_mg/dL': 0.4 // Approximate conversion
}
};

try {
  // Create conversion key
  const conversionKey = `${fromUnit}_to_${toUnit}`;

  // Get factor for the specific type
  const factor = factors[valueType]?.[conversionKey];

  if (!factor) {
    console.warn(`No conversion factor for ${valueType} from ${fromUnit} to ${toUnit}`);
    return value;
  }

  // Convert
  return value * factor;
} catch (error) {

```

```
        console.error('Error converting units:', error);
        return value;
    }
}

export default ValidationHelpers;
```

1.5 memory-manager.js

File Path: `/js/utils/memory-manager.js`

Description: Manages memory usage and optimization to prevent memory leaks and improve performance, especially for resource-intensive operations.

Features:

- Cache management with expiration
- Memory pressure handling
- Result pagination to limit memory usage
- Resource cleanup
- Cache prioritization

```javascript

```
/**
 * Memory Manager for CVD Risk Toolkit
 *
 * Manages memory usage, prevents memory leaks, and
 * implements resource cleanup for optimal performance.
 */
```

```
class MemoryManager {
 static #cachedData = new Map();
 static #maxCacheItems = 20;
 static #maxCacheAge = 30 * 60 * 1000; // 30 minutes

 /**
 * Initialize memory management
 */
 static initialize() {
 console.log('Initializing memory management...');

 // Set up periodic cleanup
 setInterval(this.cleanupCache.bind(this), 5 * 60 * 1000); // Every 5 minutes

 // Set up memory pressure handler for modern browsers
 if ('gc' in window) {
 window.addEventListener('memorypressure', this.handleMemoryPressure.bind(this));
 }

 // Initialize ResultManager for pagination
```

```

 this.resultManager = new ResultManager();
 }

 /**
 * Cache data with expiration
 * @param {string} key - Cache key
 * @param {*} data - Data to cache
 * @param {Object} options - Options
 */
 static cacheData(key, data, {
 expiration = this.#maxCacheAge,
 priority = 0
 } = {}) {
 // Enforce cache size limit
 if (this.#cachedData.size >= this.#maxCacheItems) {
 this.cleanupCache();

 // If still at limit, remove oldest item
 if (this.#cachedData.size >= this.#maxCacheItems) {
 let oldestKey = null;
 let oldestTime = Infinity;

 for (const [k, v] of this.#cachedData.entries()) {
 if (v.timestamp < oldestTime && v.priority <= priority) {
 oldestTime = v.timestamp;
 oldestKey = k;
 }
 }

 if (oldestKey) {
 this.#cachedData.delete(oldestKey);
 }
 }
 }

 // Add to cache with timestamp
 this.#cachedData.set(key, {
 data,
 timestamp: Date.now(),
 expiration,
 priority
 });
 }
}

```

```
/**
 * Get cached data
 * @param {string} key - Cache key
 * @param {*} defaultValue - Default value if key doesn't exist
 * @returns {*} - Cached data or defaultValue
 */
static getCachedData(key, defaultValue = null) {
 const cached = this.#cachedData.get(key);

 if (!cached) {
 return defaultValue;
 }

 // Check if expired
 if (Date.now() - cached.timestamp > cached.expiration) {
 this.#cachedData.delete(key);
 return defaultValue;
 }

 return cached.data;
}

/**
 * Clear entire cache or specific key
 * @param {string} key - Optional key to clear
 */
static clearCache(key = null) {
 if (key) {
 this.#cachedData.delete(key);
 } else {
 this.#cachedData.clear();
 }
}

/**
 * Cleanup expired cache items
 */
static cleanupCache() {
 const now = Date.now();

 for (const [key, value] of this.#cachedData.entries()) {
 if (now - value.timestamp > value.expiration) {
 this.#cachedData.delete(key);
 }
 }
}
```

```

 }

}

/***
 * Handle memory pressure events
 * @param {Event} event - Memory pressure event
 */
static handleMemoryPressure(event) {
 console.warn(`Memory pressure detected: ${event.pressure}`);

 // Clear non-critical cache on high pressure
 if (event.pressure === 'critical') {
 this.#cachedData.clear();
 this.resultManager.clearOldResults();

 // Force garbage collection if available
 if (typeof window.gc === 'function') {
 window.gc();
 }
 } else {
 // Clean up expired items
 this.cleanupCache();
 }
}

/***
 * Add calculation result to manager
 * @param {*} result - Calculation result
 * @returns {*} - The result
 */
static addResult(result) {
 return this.resultManager.addResult(result);
}

/***
 * Get paginated results
 * @param {number} page - Page number (0-based)
 * @param {number} resultsPerPage - Results per page
 * @returns {Array} - Results for the page
 */
static getResultsForPage(page = 0, resultsPerPage = 5) {
 return this.resultManager.getResultsForPage(page, resultsPerPage);
}
}

```

```
/**
 * Result Manager Class for paginating and managing results
 */

class ResultManager {
 constructor(maxStoredResults = 10) {
 this.results = [];
 this.maxStoredResults = maxStoredResults;
 this.currentPage = 0;
 }

 /**
 * Add a result to the manager
 * @param {*} result - Result to add
 * @returns {*} - The result
 */
 addResult(result) {
 // Add to beginning of array
 this.results.unshift(result);

 // Trim if exceeding maximum
 if (this.results.length > this.maxStoredResults) {
 this.results = this.results.slice(0, this.maxStoredResults);
 }

 // Reset to first page
 this.currentPage = 0;

 return this.results[0];
 }

 /**
 * Get results for a specific page
 * @param {number} page - Page number (0-based)
 * @param {number} resultsPerPage - Results per page
 * @returns {Array} - Results for the page
 */
 getResultsForPage(page = 0, resultsPerPage = 5) {
 const start = page * resultsPerPage;
 const end = start + resultsPerPage;
 return this.results.slice(start, end);
 }
}
```

```
* Clear all but the most recent result
*/
clearOldResults() {
 if (this.results.length > 1) {
 this.results = [this.results[0]];
 this.currentPage = 0;
 }
}

export default MemoryManager;
```

## 1.6 cross-tab-sync.js

**File Path:** /js/utils/cross-tab-sync.js

**Description:** Enables synchronization of data and state between browser tabs using localStorage and the storage event, ensuring a consistent user experience.

**Features:**

- Event-based cross-tab communication
- Form data synchronization
- Risk calculation result sharing
- Heartbeat mechanism for active tab detection
- Debounced updates to prevent race conditions

javascript

```

/**
 * Cross-Tab Synchronization for CVD Risk Toolkit
 *
 * Enables data synchronization between browser tabs using
 * localStorage and the storage event.
 */

import EventBus from './event-bus.js';

class CrossTabSync {
 static #syncKey = 'cvd-toolkit-sync';
 static #initialized = false;

 /**
 * Initialize cross-tab synchronization
 */
 static initialize() {
 if (this.#initialized) return;

 console.log('Initializing cross-tab synchronization...');

 // Listen for storage events
 window.addEventListener('storage', this.#handleStorageEvent.bind(this));

 // Create unique tab ID
 this.tabId = `tab-${Date.now()}-${Math.random().toString(36).substr(2, 9)}`;

 // Set up heartbeat to detect active tabs
 setInterval(this.#sendHeartbeat.bind(this), 5000);

 this.#initialized = true;

 // Subscribe to form changes
 EventBus.subscribe('form-field-changed', (data) => {
 this.syncData({
 tabType: data.formId.replace('-form', ''),
 field: data.fieldId,
 value: data.value,
 timestamp: Date.now()
 });
 });

 // Subscribe to risk calculations
 }

 syncData(data) {
 const key = `${this.tabId}-${

```

```

EventBus.subscribe('risk-calculated', (data) => {
 this.syncData({
 tabType: data.calculator.toLowerCase(),
 result: data,
 timestamp: Date.now()
 });
});

/**
 * Handle storage event from other tabs
 * @param {StorageEvent} event - Storage event
 */
static #handleStorageEvent(event) {
 if (event.key !== this.#syncKey) return;

 try {
 const data = JSON.parse(event.newValue);

 // Ignore events from this tab
 if (data.tabId === this.tabId) return;

 // Process based on data type
 if (data.field && data.value !== undefined) {
 // Field update
 EventBus.publish('sync-data-changed', {
 tabType: data.tabType,
 values: { [data.field]: data.value },
 timestamp: data.timestamp
 });
 }

 // Update field if present
 const element = document.getElementById(data.field);
 if (element) {
 if (element.type === 'checkbox') {
 element.checked = Boolean(data.value);
 } else {
 element.value = data.value;
 }
 }

 // Trigger change event
 element.dispatchEvent(new Event('change', { bubbles: true }));
 }
} else if (data.result) {
}

```

```

 // Risk calculation result
 EventBus.publish('sync-result-available', {
 tabType: data.tabType,
 result: data.result,
 timestamp: data.timestamp
 });
 } else if (data.heartbeat) {
 // Heartbeat from another tab
 EventBus.publish('tab-heartbeat', {
 tabId: data.tabId,
 timestamp: data.timestamp
 });
 }
} catch (error) {
 console.error('Error handling storage event:', error);
}
}

/**
 * Synchronize data with other tabs
 * @param {Object} data - Data to synchronize
 */
static syncData(data) {
 try {
 // Add tab ID and timestamp
 const syncData = {
 ...data,
 tabId: this.tabId,
 timestamp: Date.now()
 };

 // Store in LocalStorage to trigger storage event in other tabs
 localStorage.setItem(this.#syncKey, JSON.stringify(syncData));
 } catch (error) {
 console.error('Error synchronizing data:', error);
 }
}

/**
 * Send heartbeat to other tabs
 */
static #sendHeartbeat() {
 this.syncData({
 heartbeat: true,

```

```

 timestamp: Date.now()
 });
}

/**
 * Get list of active tabs
 * @returns {Array} - List of active tab IDs
 */
static getActiveTabs() {
 // Tabs are considered active if they sent a heartbeat in the last 10 seconds
 const tabs = [];
 const now = Date.now();
 const maxAge = 10000; // 10 seconds

 try {
 const syncData = JSON.parse(localStorage.getItem(this.#syncKey) || '{}');

 if (syncData.heartbeat && now - syncData.timestamp < maxAge) {
 tabs.push(syncData.tabId);
 }
 } catch (error) {
 console.error('Error getting active tabs:', error);
 }

 // Always include this tab
 if (!tabs.includes(this.tabId)) {
 tabs.push(this.tabId);
 }

 return tabs;
}
}

export default CrossTabSync;

```

## 1.7 tab-manager.js

**File Path:** /js/utils/tab-manager.js

**Description:** Manages tab navigation, content switching, and tab state persistence for the application's tabbed interface.

**Features:**

- Tab navigation with proper event handling
- Content switching with animations
- Tab state persistence
- Accessibility support with ARIA attributes
- URL hash synchronization for bookmarking
- Event publication for tab changes

javascript

```

/**
 * Tab Manager for CVD Risk Toolkit
 *
 * Manages tab navigation, content switching, and tab state persistence.
 */

import EventBus from './event-bus.js';

class TabManager {
 #tabs = [];
 #activeTab = null;
 #options = {};
 #initialized = false;

 /**
 * Create a new TabManager instance
 * @param {Object} options - Options
 */
 constructor(options = {}) {
 this.#options = {
 tabSelector: '.tab',
 contentSelector: '.tab-content',
 activeClass: 'active',
 onChange: null,
 persistState: true,
 ...options
 };
 this.#initialize();
 }

 /**
 * Initialize tab manager
 */
 #initialize() {
 if (this.#initialized) return;

 console.log('Initializing tab manager...');

 // Find all tabs
 this.#tabs = Array.from(document.querySelectorAll(this.#options.tabSelector));

 // Set up event listeners
 }
}

```

```

this.#tabs.forEach(tab => {
 tab.addEventListener('click', this.#handleTabClick.bind(this));
});

// Restore active tab from session storage if enabled
if (this.#options.persistState) {
 const activeTabId = sessionStorage.getItem('active-tab');

 if (activeTabId) {
 const tab = this.#tabs.find(t => t.id === activeTabId || t.getAttribute('data-t

 if (tab) {
 this.activateTab(tab);
 return;
 }
 }
}

// Activate first tab by default if none is active
const activeTab = this.#tabs.find(tab => tab.classList.contains(this.#options.activeClas

if (activeTab) {
 this.activateTab(activeTab);
} else if (this.#tabs.length > 0) {
 this.activateTab(this.#tabs[0]);
}

// Mark as initialized
this.#initialized = true;
window.__tabManagerLoaded = true;

// Listen for hash changes for direct tab navigation
window.addEventListener('hashchange', this.#handleHashChange.bind(this));

// Handle initial hash
this.#handleHashChange();
}

/**
 * Handle tab click event
 * @param {Event} event - Click event
 */
#handleTabClick(event) {
 event.preventDefault();
}

```

```

 this.activateTab(event.currentTarget);
 }

 /**
 * Handle hash change event for direct tab navigation
 */
 #handleHashChange() {
 const hash = window.location.hash.substring(1);

 if (hash) {
 // Find tab by data-tab attribute or matching content ID
 const tab = this.#tabs.find(
 t => t.getAttribute('data-tab') === hash ||
 t.getAttribute('aria-controls') === `content-${hash}`
);
 }

 if (tab) {
 this.activateTab(tab);
 }
 }
}

/**
 * Activate a tab
 * @param {Element|string} tab - Tab element or ID/data-tab value
 */
activateTab(tab) {
 // If string provided, find matching tab
 if (typeof tab === 'string') {
 const tabElement = this.#tabs.find(
 t => t.id === tab ||
 t.getAttribute('data-tab') === tab ||
 t.getAttribute('aria-controls') === `content-${tab}`
);
 }

 if (!tabElement) {
 console.error(`Tab not found: ${tab}`);
 return;
 }

 tab = tabElement;
}

// Skip if already active

```

```
if (this.#activeTab === tab) return;

// Get tab and content data
const tabId = tab.getAttribute('data-tab');
const controlId = tab.getAttribute('aria-controls') || `content-${tabId}`;
const content = document.getElementById(controlId);

if (!content) {
 console.error(`Tab content not found for tab: ${tabId}`);
 return;
}

// Deactivate all tabs and content
this.#tabs.forEach(t => {
 t.classList.remove(this.#options.activeClass);
 t.setAttribute('aria-selected', 'false');

 const tContentId = t.getAttribute('aria-controls') || `content-${t.getAttribute('da
 const tContent = document.getElementById(tContentId);

 if (tContent) {
 tContent.classList.remove(this.#options.activeClass);
 tContent.setAttribute('hidden', '');
 }
});

// Activate selected tab and content
tab.classList.add(this.#options.activeClass);
tab.setAttribute('aria-selected', 'true');
content.classList.add(this.#options.activeClass);
content.removeAttribute('hidden');

// Update active tab reference
this.#activeTab = tab;

// Persist state if enabled
if (this.#options.persistState) {
 sessionStorage.setItem('active-tab', tabId);
}

// Update URL hash for bookmarking
// Use history API to avoid page jumping
const url = new URL(window.location);
url.hash = tabId;
```

```

 window.history.pushState({}, '', url);

 // Call onChange callback if provided
 if (this.#options.onChange) {
 const index = this.#tabs.indexOf(tab);
 this.#options.onChange(index, tab, content);
 }

 // Publish tab change event
 EventBus.publish('tab-changed', {
 tabId,
 contentId,
 index: this.#tabs.indexOf(tab)
 });
 }

 /**
 * Get active tab
 * @returns {Element} - Active tab
 */
 getActiveTab() {
 return this.#activeTab;
 }

 /**
 * Get active tab index
 * @returns {number} - Active tab index
 */
 getActiveTabIndex() {
 return this.#tabs.indexOf(this.#activeTab);
 }

 /**
 * Activate next tab
 */
 nextTab() {
 const currentIndex = this.getActiveTabIndex();
 const nextIndex = (currentIndex + 1) % this.#tabs.length;
 this.activateTab(this.#tabs[nextIndex]);
 }

 /**
 * Activate previous tab
 */

```

```
prevTab() {
 const currentIndex = this.getActiveTabIndex();
 const prevIndex = (currentIndex - 1 + this.tabs.length) % this.tabs.length;
 this.activateTab(this.tabs[prevIndex]);
}
}
```

// Static method for use without instantiation

```
TabManager.activateTab = function(tabId) {
 const tab = document.querySelector(`.tab[data-tab="${tabId}"]`) ||
 document.querySelector(`#tab-${tabId}`);
 # Enhanced CVD Risk Toolkit - Implementation Guide
```

## ## Overview

The Enhanced CVD Risk Toolkit is a comprehensive web application designed **for** healthcare profes

### #### Purpose and Goals

1. **Clinical Decision Support**: Provide healthcare professionals **with** accurate cardiovascular risk assessments.
2. **Evidence-Based Recommendations**: Generate treatment recommendations based on the latest clinical guidelines.
3. **Visual Risk Communication**: Facilitate patient education and shared decision-making through clear, graphical displays.
4. **Clinical Workflow Integration**: Support integration **with** electronic medical records (EMR).
5. **Accessibility and Performance**: Ensure the toolkit functions effectively across various devices and user needs.

### ### Key Features and Enhancements

#### #### Core Functionality

- Multiple evidence-based risk **calculators** (Framingham, QRISK3)
- Side-by-side risk comparison visualization
- Treatment recommendations based on Canadian guidelines
- Advanced visualizations **for** risk factor analysis
- Secure data **import/export with** encryption
- EMR/FHIR integration capabilities
- Cross-browser and mobile compatibility
- Offline functionality

#### #### Clinical Algorithm Improvements

- Enhanced edge **case** handling **for** extreme clinical scenarios
- Added physiologically plausible value checks **with** warnings

- Updated clinical validation information
- Added clearer documentation **of** risk categorization thresholds
- Added specific guideline references **with** badges

#### #### Security Enhancements

- Implemented Content Security Policy
- Added encrypted local storage **for** saved calculations
- Added robust input sanitization to prevent **XSS**
- Enhanced form submission security

#### #### UI/UX Improvements

- Added visually pleasing loading indicators
- Improved cross-browser compatibility
- Enhanced mobile responsiveness
- Made legal disclaimer more prominent
- Improved date picker components
- Fixed keyboard submission validation issues
- Enhanced PDF **export** and preview

#### #### Validation & Error Handling

- Added warnings **for** implausible values
- Improved error messages and display
- Enhanced client-side validation
- Added **null** reference protection
- Added division by zero protection

### ## Module Directory Structure

This implementation guide includes detailed code and specifications **for** all required modules:

#### #### 1. Core Utilities

- `./js/utils/error-detection-system.js` - **Error** tracking and user notifications
- `./js/utils/runtime-protection.js` - **Error** handling and **performance** optimization
- `./js/utils/event-bus.js` - Centralized event publish/subscribe system
- `./js/utils/validation-helpers.js` - Clinical data validation and unit conversions
- `./js/utils/memory-manager.js` - Memory optimization and resource management
- `./js/utils/cross-tab-sync.js` - Data synchronization between browser tabs
- `./js/utils/tab-manager.js` - Tab navigation and content switching
- `./js/utils/device-capability-detector.js` - Device capability detection
- `./js/utils/module-loader.js` - Dynamic module loading
- `./js/utils/batch-processor.js` - Batch operations and concurrency management
- `./js/utils/secure-storage.js` - Encrypted local storage
- `./js/utils/xss-protection.js` - Input sanitization and **XSS** prevention

## #### 2. Calculation Algorithms

- `~/js/calculations/framingham-algorithm.js` - Framingham Risk Score implementation
- `~/js/calculations/qrisk3-algorithm.js` - QRISK3 algorithm implementation
- `~/js/calculations/treatment-recommendations.js` - Evidence-based treatment suggestions

## #### 3. Visualization Components

- `~/js/visualizations/chart-renderer.js` - Chart generation and rendering
- `~/js/visualizations/chart-exporter.js` - Chart export functionality
- `~/js/visualizations/combined-view-manager.js` - Multi-calculator view management

## #### 4. Data Management

- `~/js/data-management/data-import-export.js` - Data import/export functionality
- `~/js/data-management/emr-connector.js` - EMR system integration
- `~/js/data-management/field-mapper.js` - Data field mapping
- `~/js/data-management/pdf-generator.js` - PDF report generation
- `~/js/data-management/loading-manager.js` - Loading indicator management

## #### 5. Core Application Files

- `~/js/main.js` - Main application initialization and control
- `~/js/ui.js` - User interface interactions
- `~/js/service-worker.js` - Offline functionality
- `~/css/styles.css` - Application styling
- `~/js/form-enhancements.js` - Advanced form handling
- `~/js/disclaimer-enhancer.js` - Legal disclaimer management
- `~/js/results-display.js` - Treatment recommendations display

## ## File Modifications

### #### Modified Files

- \*\*index.html\*\*: Added security headers, loading templates, script references
- \*\*styles.css\*\*: Added enhanced styling, mobile optimizations, cross-browser fixes
- \*\*calculations.js\*\*: Enhanced clinical validation, documentation, and edge case handling
- \*\*validation.js\*\*: Added physiological validation and improved error handling

### #### New Files

- \*\*js/loading-manager.js\*\*: Handles loading indicators during `async` operations
- \*\*js/secure-storage.js\*\*: Provides encrypted local storage
- \*\*js/xss-protection.js\*\*: Input sanitization and XSS prevention
- \*\*js/form-enhancements.js\*\*: Improved form handling and keyboard navigation
- \*\*js/pdf-enhancer.js\*\*: Enhanced PDF export functionality
- \*\*js/disclaimer-enhancer.js\*\*: Improved legal disclaimer
- \*\*js/results-display.js\*\*: Enhanced treatment recommendations display

## ## Implementation Details

Each module is described below **with** complete code implementation to allow another **AI** to **continue**

---

## 1. Core Utilities### 1.1 error-detection-system.js

**File Path**: `/js/utils/error-detection-system.js`

**Description**: Central error handling system that captures, logs, and displays user-friendly

**Features**:

- Global error event capturing
- Unhandled promise rejection handling
- User-friendly error notifications
- Critical error page fallback
- Error categorization and prioritization
- Integration with analytics services
- Online/offline state notifications

```javascript

```
/**  
 * Error Detection System for CVD Risk Toolkit  
 *  
 * Provides centralized error handling, logging, and user notifications  
 * for application errors.  
 */
```

```
class ErrorDetectionSystem {  
    static #errors = [];  
    static #notificationContainer = null;  
  
    /**  
     * Initialize error tracking  
     */  
    static initErrorTracking() {  
        console.log('Initializing error tracking system...');  
  
        // Set up global error handler  
        window.addEventListener('error', this.#handleGlobalError.bind(this));  
        window.addEventListener('unhandledrejection', this.#handleUnhandledRejection.bind(this));  
  
        // Find notification container  
        document.addEventListener('DOMContentLoaded', () => {
```

```
        this.#notificationContainer = document.getElementById('error-notification-container')
        if (!this.#notificationContainer) {
            console.warn('Error notification container not found');
        }
    });

}

/***
 * Handle global error event
 * @param {ErrorEvent} event - Error event
 */
static #handleGlobalError(event) {
    this.trackError({
        message: event.message,
        source: event.filename,
        lineno: event.lineno,
        colno: event.colno,
        error: event.error
    });
}

/***
 * Handle unhandled Promise rejection
 * @param {PromiseRejectionEvent} event - Rejection event
 */
static #handleUnhandledRejection(event) {
    this.trackError({
        message: event.reason?.message || 'Unhandled Promise rejection',
        error: event.reason,
        type: 'promise'
    });
}

/***
 * Track an error
 * @param {Object} errorInfo - Error information
 */
static trackError(errorInfo) {
    this.#errors.push({
        ...errorInfo,
        timestamp: new Date()
    });

    console.error('Tracked error:', errorInfo);
}
```

```
// Log to analytics or error tracking service if available
if (window.errorTrackingService) {
    window.errorTrackingService.logError(errorInfo);
}

/**
 * Show error notification to user
 * @param {Error} error - Error object
 * @param {string} title - Error title
 */
static showErrorNotification(error, title = 'Error') {
    if (!this.#notificationContainer) {
        this.#notificationContainer = document.getElementById('error-notification-container')
        if (!this.#notificationContainer) {
            console.error('Error notification container not found');
            return;
        }
    }
}

const notification = document.createElement('div');
notification.className = 'error-notification';
notification.innerHTML =
    `<div class="notification-header">
        <span class="notification-title">${title}</span>
        <button class="notification-close" aria-label="Close notification">&times;</but
    </div>
    <div class="notification-body">
        <p>${error.message || 'An error occurred'}</p>
    </div>
`;

// Add to notification container
this.#notificationContainer.appendChild(notification);

// Show with animation
setTimeout(() => notification.classList.add('show'), 10);

// Add event listener for close button
notification.querySelector('.notification-close')?.addEventListener('click', () => {
    notification.classList.remove('show');
    setTimeout(() => notification.remove(), 300);
});
```

```

// Auto-hide after 5 seconds
setTimeout(() => {
    notification.classList.remove('show');
    setTimeout(() => notification.remove(), 300);
}, 5000);
}

/**
 * Show critical error page
 * @param {Error} error - Error object
 */
static showCriticalErrorPage(error) {
    // Track the error
    this.trackError({
        message: error.message,
        error: error,
        type: 'critical'
    });

    // Replace page content with error page
    document.body.innerHTML = `
        <div class="critical-error-page">
            <div class="error-icon">⚠</div>
            <h1>Application Error</h1>
            <p>We're sorry, but a critical error has occurred:</p>
            <div class="error-message">${error.message || 'Unknown error'}</div>
            <p>Please try refreshing the page. If the problem persists, contact support.</p>
            <button class="btn btn-primary" onclick="location.reload()">Refresh Page</button>
        </div>
    `;
}

/**
 * Handle module loading errors
 * @param {Array} failedModules - List of failed module loads
 */
static handleModuleLoadingErrors(failedModules) {
    if (!failedModules || failedModules.length === 0) return;

    const moduleNames = failedModules.map(m => {
        return m.reason?.moduleName || m.reason?._modulePath?.split('/').pop() || 'unknown';
    }).join(', ');
}

```

```
        this.showErrorNotification(
            new Error(`Failed to load modules: ${moduleNames}. Some features may be unavailable
            'Module Loading Error'
        );
    }

    /**
     * Show online notification
     */
    static showOnlineNotification() {
        if (!this.#notificationContainer) {
            this.#notificationContainer = document.getElementById('error-notification-container');
            if (!this.#notificationContainer) return;
        }

        const notification = document.createElement('div');
        notification.className = 'success-notification';
        notification.innerHTML =
            `<div class="notification-header">
                <span class="notification-title">Online</span>
                <button class="notification-close" aria-label="Close notification">&times;</button>
            </div>
            <div class="notification-body">
                <p>Connection restored. All features are now available.</p>
            </div>
        `;
        // Add to notification container
        this.#notificationContainer.appendChild(notification);

        // Show with animation
        setTimeout(() => notification.classList.add('show'), 10);

        // Add event listener for close button
        notification.querySelector('.notification-close')?.addEventListener('click', () => {
            notification.classList.remove('show');
            setTimeout(() => notification.remove(), 300);
        });

        // Auto-hide after 3 seconds
        setTimeout(() => {
            notification.classList.remove('show');
            setTimeout(() => notification.remove(), 300);
        }, 3000);
    }
}
```

```

}

/**
 * Show offline notification
 */
static showOfflineNotification() {
    if (!this.#notificationContainer) {
        this.#notificationContainer = document.getElementById('error-notification-container');
        if (!this.#notificationContainer) return;
    }

    const notification = document.createElement('div');
    notification.className = 'warning-notification';
    notification.innerHTML =
        `<div class="notification-header">
            <span class="notification-title">Offline</span>
            <button class="notification-close" aria-label="Close notification">&times;</button>
        </div>
        <div class="notification-body">
            <p>You are currently offline. Some features may be unavailable.</p>
        </div>
`;
}

// Add to notification container
this.#notificationContainer.appendChild(notification);

// Show with animation
setTimeout(() => notification.classList.add('show'), 10);

// Add event listener for close button
notification.querySelector('.notification-close')?.addEventListener('click', () => {
    notification.classList.remove('show');
    setTimeout(() => notification.remove(), 300);
});

}

}

export default ErrorDetectionSystem;

```

1.2 runtime-protection.js

File Path: /js/utils/runtime-protection.js

Description: Provides utility functions for error handling, performance optimization, and runtime protection. This module helps prevent common JavaScript errors and improves performance through throttling and debouncing.

Features:

- Safe error handling with tryCatch
- Function debouncing for performance
- Function throttling for rate limiting
- Safe JSON parsing and stringifying
- Safely accessing nested object properties
- Retry mechanism for fallible operations

javascript

```

/**
 * Runtime Protection for CVD Risk Toolkit
 *
 * Provides error handling, debouncing, throttling, and other runtime
 * protection utilities.
 */

class RuntimeProtection {
    /**
     * Execute a function with error handling
     * @param {Function} fn - Function to execute
     * @param {Function} errorHandler - Error handler function
     * @returns {*} - Result of function or error handler
     */
    static tryCatch(fn, errorHandler) {
        try {
            return fn();
        } catch (error) {
            console.error('Runtime protection caught error:', error);
            return errorHandler ? errorHandler(error) : null;
        }
    }

    /**
     * Debounce a function
     * @param {Function} fn - Function to debounce
     * @param {number} delay - Delay in milliseconds
     * @returns {Function} - Debounced function
     */
    static debounce(fn, delay = 300) {
        let timeoutId;

        return function(...args) {
            clearTimeout(timeoutId);
            timeoutId = setTimeout(() => fn.apply(this, args), delay);
        };
    }

    /**
     * Throttle a function
     * @param {Function} fn - Function to throttle
     * @param {number} delay - Delay in milliseconds
     * @returns {Function} - Throttled function
     */
}

```

```

/*
static throttle(fn, delay = 300) {
    let lastCall = 0;
    let timeoutId = null;

    return function(...args) {
        const now = Date.now();
        const context = this;

        if (now - lastCall >= delay) {
            lastCall = now;
            return fn.apply(context, args);
        } else {
            // Clear any existing timeout
            clearTimeout(timeoutId);

            // Set a new timeout
            timeoutId = setTimeout(() => {
                lastCall = Date.now();
                fn.apply(context, args);
            }, delay - (now - lastCall));
        }
    };
}

/**
 * Safely parse JSON
 * @param {string} json - JSON string
 * @param {*} fallback - Fallback value
 * @returns {*} - Parsed JSON or fallback
 */
static safeParseJSON(json, fallback = {}) {
    if (!json) return fallback;

    try {
        return JSON.parse(json);
    } catch (error) {
        console.error('Error parsing JSON:', error);
        return fallback;
    }
}

/**
 * Safely stringify JSON

```

```

* @param {*} value - Value to stringify
* @param {*} fallback - Fallback value
* @returns {string} - JSON string or fallback
*/
static safeStringifyJSON(value, fallback = '{}') {
    try {
        return JSON.stringify(value);
    } catch (error) {
        console.error('Error stringifying JSON:', error);
        return fallback;
    }
}

/**
 * Safely access nested object properties
* @param {Object} obj - Object to access
* @param {string} path - Property path (e.g., 'a.b.c')
* @param {*} fallback - Fallback value
* @returns {*} - Property value or fallback
*/
static safeGet(obj, path, fallback = null) {
    if (!obj || !path) return fallback;

    const keys = path.split('.');
    let result = obj;

    for (const key of keys) {
        if (result === null || result === undefined) {
            return fallback;
        }

        result = result[key];
    }

    return result !== undefined ? result : fallback;
}

/**
 * Create a safe function that catches errors
* @param {Function} fn - Function to make safe
* @param {*} fallbackValue - Value to return on error
* @returns {Function} - Safe function
*/
static safeFunction(fn, fallbackValue = null) {

```

```

        return function(...args) {
            try {
                return fn.apply(this, args);
            } catch (error) {
                console.error('Error in safe function:', error);
                return fallbackValue;
            }
        };
    }

    /**
     * Retry a function multiple times
     * @param {Function} fn - Function to retry
     * @param {Object} options - Options
     * @returns {Promise} - Promise resolving to function result
     */
    static async retry(fn, {
        retries = 3,
        delay = 500,
        backoff = 2,
        onRetry = null
    } = {}) {
        let lastError;

        for (let attempt = 0; attempt <= retries; attempt++) {
            try {
                return await fn();
            } catch (error) {
                lastError = error;

                if (attempt < retries) {
                    const delayTime = delay * Math.pow(backoff, attempt);

                    if (onRetry) {
                        onRetry(error, attempt, delayTime);
                    }
                }

                await new Promise(resolve => setTimeout(resolve, delayTime));
            }
        }
    }

    throw lastError;
}

```

```
}
```

```
export default RuntimeProtection;
```

1.3 event-bus.js

File Path: /js/utils/event-bus.js

Description: Implements a centralized event bus for publishing and subscribing to events throughout the application. Provides a decoupled communication mechanism between components.

Features:

- Event subscription with callbacks
- Event publishing with data
- Subscription management with unsubscribe capability
- Error handling for subscriber callbacks
- Subscription status checking

javascript

```

/**
 * Event Bus for CVD Risk Toolkit
 *
 * Provides a centralized event bus for publishing and subscribing to events.
 */

class EventBus {
    static #subscribers = {};

    /**
     * Subscribe to an event
     * @param {string} event - Event name
     * @param {Function} callback - Callback function
     * @returns {Object} - Subscription with unsubscribe method
     */
    static subscribe(event, callback) {
        if (!this.#subscribers[event]) {
            this.#subscribers[event] = [];
        }

        const index = this.#subscribers[event].length;
        this.#subscribers[event].push(callback);

        // Return subscription object with unsubscribe method
        return {
            unsubscribe: () => {
                if (this.#subscribers[event] && this.#subscribers[event][index]) {
                    this.#subscribers[event][index] = null;
                }
            }
        };
    }

    /**
     * Publish an event
     * @param {string} event - Event name
     * @param {*} data - Event data
     */
    static publish(event, data) {
        if (!this.#subscribers[event]) {
            return;
        }
    }
}

```

```

// Execute callbacks and filter out null entries (unsubscribed)
this.#subscribers[event] = this.#subscribers[event]
    .filter(callback => callback !== null)
    .filter(callback => {
        try {
            callback(data);
            return true;
        } catch (error) {
            console.error(`Error in event subscriber for "${event}":`, error);
            return false;
        }
    });
}

/**
 * Check if an event has subscribers
 * @param {string} event - Event name
 * @returns {boolean} - Whether event has subscribers
 */
static hasSubscribers(event) {
    return this.#subscribers[event] &&
        this.#subscribers[event].filter(cb => cb !== null).length > 0;
}

/**
 * Get number of subscribers for an event
 * @param {string} event - Event name
 * @returns {number} - Number of subscribers
 */
static subscriberCount(event) {
    if (!this.#subscribers[event]) {
        return 0;
    }

    return this.#subscribers[event].filter(cb => cb !== null).length;
}

/**
 * Reset all event subscriptions
 */
static reset() {
    this.#subscribers = {};
}

```

```
/**  
 * Reset specific event subscriptions  
 * @param {string} event - Event name  
 */  
  
static resetEvent(event) {  
    this.#subscribers[event] = [];  
}  
  
}  
  
export default EventBus;
```

1.4 validation-helpers.js

File Path: `/js/utils/validation-helpers.js`

Description: Provides functions for validating clinical data, calculating derived values, and performing unit conversions. Ensures data integrity and physiological plausibility.

Features:

- Clinical value validation against reference ranges
- Outlier detection for implausible values
- Derived value calculations (BMI, LDL, etc.)
- Unit conversions between measurement systems
- Standard deviation calculations for repeated measures

javascript

```

/**
 * Validation Helpers for CVD Risk Toolkit
 *
 * Provides functions for validating clinical data,
 * calculating derived values, and unit conversions.
 */

class ValidationHelpers {
    // Reference ranges for clinical values
    static #referenceRanges = {
        age: { min: 18, max: 110, outlierMin: 25, outlierMax: 90 },
        height: { min: 100, max: 250, outlierMin: 140, outlierMax: 210, unit: 'cm' }, // cm
        weight: { min: 20, max: 300, outlierMin: 40, outlierMax: 160, unit: 'kg' }, // kg
        bmi: { min: 10, max: 80, outlierMin: 18.5, outlierMax: 40 },
        systolicBP: { min: 60, max: 300, outlierMin: 90, outlierMax: 200, unit: 'mmHg' },
        diastolicBP: { min: 40, max: 180, outlierMin: 60, outlierMax: 120, unit: 'mmHg' },
        totalCholesterol: {
            min: 1.0, max: 20.0, outlierMin: 2.5, outlierMax: 8.0, unit: 'mmol/L',
            minMgdL: 40, maxMgdL: 800, outlierMinMgdL: 100, outlierMaxMgdL: 300
        },
        hdl: {
            min: 0.1, max: 5.0, outlierMin: 0.8, outlierMax: 3.0, unit: 'mmol/L',
            minMgdL: 5, maxMgdL: 200, outlierMinMgdL: 30, outlierMaxMgdL: 100
        },
        ldl: {
            min: 0.5, max: 15.0, outlierMin: 1.5, outlierMax: 6.0, unit: 'mmol/L',
            minMgdL: 20, maxMgdL: 600, outlierMinMgdL: 60, outlierMaxMgdL: 240
        },
        triglycerides: {
            min: 0.2, max: 20.0, outlierMin: 0.5, outlierMax: 5.0, unit: 'mmol/L',
            minMgdL: 20, maxMgdL: 1800, outlierMinMgdL: 50, outlierMaxMgdL: 400
        },
        lipoproteinA: {
            min: 0, max: 500, outlierMin: 10, outlierMax: 200, unit: 'mg/dL',
            minNmoll: 0, maxNmoll: 1000, outlierMinNmoll: 20, outlierMaxNmoll: 400
        },
        apoB: {
            min: 0.1, max: 5.0, outlierMin: 0.5, outlierMax: 1.8, unit: 'g/L',
            minMgdL: 10, maxMgdL: 500, outlierMinMgdL: 50, outlierMaxMgdL: 180
        }
    };
}

/**

```

```

* Validate a clinical value against reference ranges
* @param {number} value - Value to validate
* @param {string} type - Type of value (e.g. 'age', 'systolicBP')
* @param {Object} options - Options
* @returns {Object} - Validation result
*/
static validateClinicalValue(value, type, {
    allowOutliers = true,
    unit = null
} = {}) {
    if (isNaN(value)) {
        return {
            isValid: false,
            value: null,
            message: 'Value must be a number'
        };
    }

    // Get reference range
    const range = this.#referenceRanges[type];
    if (!range) {
        return {
            isValid: true,
            value,
            message: 'No validation available for this type'
        };
    }

    // Determine min/max based on unit if provided
    let min = range.min;
    let max = range.max;
    let outlierMin = range.outlierMin;
    let outlierMax = range.outlierMax;

    // Handle unit-specific ranges for lipid values
    if (unit) {
        if (unit === 'mg/dL' && type.match(/cholesterol|hdl|ldl|triglycerides/)) {
            min = range.minMgdL;
            max = range.maxMgdL;
            outlierMin = range.outlierMinMgdL;
            outlierMax = range.outlierMaxMgdL;
        } else if (unit === 'nmol/L' && type === 'lipoproteinA') {
            min = range.minNmoll;
            max = range.maxNmoll;
        }
    }
}

```

```

        outlierMin = range.outlierMinNmolL;
        outlierMax = range.outlierMaxNmolL;
    } else if (unit === 'mg/dL' && type === 'apoB') {
        min = range.minMgdL;
        max = range.maxMgdL;
        outlierMin = range.outlierMinMgdL;
        outlierMax = range.outlierMaxMgdL;
    }
}

// Check if value is within range
if (value < min || value > max) {
    return {
        isValid: false,
        value: null,
        message: `Value must be between ${min} and ${max} ${range.unit} || ''`
    };
}

// Check if value is an outlier
if (value < outlierMin || value > outlierMax) {
    return {
        isValid: allowOutliers,
        isOutlier: true,
        value,
        message: `Unusual value. Common range is ${outlierMin} to ${outlierMax} ${range.unit} || ''`
    };
}

return {
    isValid: true,
    value,
    message: ''
};
}

/**
 * Calculate BMI from height and weight
 * @param {number} height - Height
 * @param {string} heightUnit - Height unit ('cm' or 'in')
 * @param {number} weight - Weight
 * @param {string} weightUnit - Weight unit ('kg' or 'lb')
 * @returns {number|null} - BMI or null if invalid inputs
 */

```

```

static calculateBMI(height, heightUnit, weight, weightUnit) {
    if (!height || !weight) {
        return null;
    }

    try {
        // Convert height to meters
        let heightM;
        if (heightUnit === 'in') {
            heightM = height * 0.0254;
        } else {
            heightM = height / 100;
        }

        // Convert weight to kg
        let weightKg;
        if (weightUnit === 'lb') {
            weightKg = weight * 0.453592;
        } else {
            weightKg = weight;
        }

        // Calculate BMI
        const bmi = weightKg / (heightM * heightM);

        // Round to 1 decimal place
        return Math.round(bmi * 10) / 10;
    } catch (error) {
        console.error('Error calculating BMI:', error);
        return null;
    }
}

/***
 * Calculate LDL using Friedewald formula
 * @param {number} tc - Total cholesterol
 * @param {number} hdl - HDL cholesterol
 * @param {number} trig - Triglycerides
 * @param {string} unit - Unit ('mmol/L' or 'mg/dL')
 * @returns {number|null} - LDL or null if invalid inputs
 */
static calculateLDL(tc, hdl, trig, unit) {
    if (!tc || !hdl || !trig) {
        return null;
}

```

```

}

try {
  let ldl;

  // Friedewald formula
  if (unit === 'mmol/L') {
    // For mmol/L: LDL = TC - HDL - (TG / 2.2)
    ldl = tc - hdl - (trig / 2.2);
  } else {
    // For mg/dL: LDL = TC - HDL - (TG / 5)
    ldl = tc - hdl - (trig / 5);
  }

  // Round to 2 decimal places
  # Enhanced CVD Risk Toolkit - Implementation Guide
}

```

Overview

The Enhanced CVD Risk Toolkit is a comprehensive web application designed **for** healthcare profes

Purpose and Goals

1. **Clinical Decision Support**: Provide healthcare professionals **with** accurate cardiovascular risk assessment tools.
2. **Evidence-Based Recommendations**: Generate treatment recommendations based on the latest clinical guidelines and research.
3. **Visual Risk Communication**: Facilitate patient education and shared decision-making through clear, graphical displays of risk information.
4. **Clinical Workflow Integration**: Support integration **with** electronic medical records (EMR) and other healthcare systems.
5. **Accessibility and Performance**: Ensure the toolkit functions effectively across various devices and user interfaces.

Key Features

1. **Multiple Risk Calculators**:
 - Framingham Risk **Score** (**2008** algorithm) **with** accurate coefficient implementation
 - QRISK3 (**2017** algorithm) **with** full risk factor support
 - Side-by-side comparison **of** results **for** comprehensive risk assessment
2. **Clinical Recommendations**:
 - Treatment recommendations following Canadian Cardiovascular Society **Guidelines** (**2021**)
 - BC PharmaCare Special Authority criteria **for** PCSK9 inhibitors
 - Medication effectiveness evaluation and target setting based on risk category

3. **Advanced Visualizations**:

- Risk progression over time projections
- Risk factor impact analysis showing contribution **of** modifiable factors
- Sensitivity analysis **for** understanding parameter influence
- Treatment effect projections to visualize intervention benefits

4. **Data Management**:

- Import/**export** functionality **with** support **for** multiple formats
- **EMR/FHIR** integration **for** clinical workflow enhancement
- Cross-tab data synchronization **for** multi-view analysis
- Secure local storage **with** encryption options

5. **Progressive Enhancement**:

- Device capability detection **for** optimized **performance**
- Performance adjustments **for** low-end devices
- Offline functionality **for** reliability **in** clinical settings
- Accessibility features **for** diverse user needs

Applied Enhancements

Clinical Algorithm Improvements

- Enhanced edge **case** handling **for** extreme clinical scenarios
- Added physiologically plausible value checks **with** warnings
- Updated clinical validation information
- Added clearer documentation **of** risk categorization thresholds
- Added specific guideline references **with** badges

Security Enhancements

- Implemented Content Security Policy
- Added encrypted local storage **for** saved calculations
- Added robust input sanitization to prevent **XSS**
- Enhanced form submission security

UI/UX Improvements

- Added visually pleasing loading indicators
- Improved cross-browser compatibility
- Enhanced mobile responsiveness
- Made legal disclaimer more prominent
- Improved date picker components
- Fixed keyboard submission validation issues
- Enhanced **PDF export** and preview

Validation & Error Handling

- Added warnings `for` implausible values
- Improved error messages and display
- Enhanced client-side validation
- Added `null` reference protection
- Added division by zero protection

Module Overview

This implementation guide provides detailed code and specifications `for` the following modules:

1. **Core Utilities - Foundation components `for` application functionality:**

- `Error` handling and detection
- Runtime protection and optimization
- Event management
- Data validation
- Memory management
- Cross-tab synchronization
- `UI` component management
- Device capability detection
- Module loading
- Batch processing
- Secure storage
- `XSS` protection

2. **Calculation Algorithms - Evidence-based risk calculation implementations:**

- Framingham Risk Score algorithm
- `QRISK3` algorithm
- Treatment recommendations generator

3. **Visualization Components - Interactive data visualization tools:**

- Chart rendering engine
- Chart `export` functionality
- Combined view management

4. **Data Management - Data handling and integration capabilities:**

- Data `import/export`
- `EMR` connector
- Field mapper
- `PDF` generation

5. **Core Application Files - Main application structure:**

- Main application controller
- `UI` interaction management
- Service worker `for` offline functionality

- CSS styling

Implementation Details

Each module is described **in detail** **with** complete code implementation to allow another **AI** to **correct**

1. Core Utilities// Fetch event handler

```
self.addEventListener('fetch', event => {
    // Skip cross-origin requests
    if (!event.request.url.startsWith(self.location.origin)) {
        return;
    }

    // Skip URLs with query params (API requests)
    if (event.request.url.includes('?')) {
        return;
    }

    // Handle fetch event
    event.respondWith(
        caches.match(event.request)
            .then(cachedResponse => {
                // Return cached response if available
                if (cachedResponse) {
                    return cachedResponse;
                }

                // Otherwise fetch from network
                return fetch(event.request)
                    .then(response => {
                        // Cache response for future
                        return caches.open(CACHE_NAMES.dynamic)
                            .then(cache => {
                                // Clone response to cache and return
                                if (response.status === 200) {
                                    cache.put(event.request, response.clone());
                                }
                                return response;
                            });
                    })
                    .catch(error => {
                        // Network error, fallback to offline page
                    });
            })
            .catch(error => {
                // Network error, fallback to offline page
            });
    );
});
```

```
        console.error('[Service Worker] Fetch error:', error);

        // Check if request is for HTML page
        if (event.request.headers.get('accept').includes('text/html')) {
            return caches.match('/offline.html');
        }

        // No fallback for other requests
        return new Response('Network error', { status: 503, statusText: 'Service
    });

}

// Message event handler
self.addEventListener('message', event => {
    console.log('[Service Worker] Message received:', event.data);

    // Handle messages from client
    if (event.data && event.data.type) {
        switch (event.data.type) {
            case 'CACHE_NEW_ASSET':
                // Cache new asset
                cacheAsset(event.data.url);
                break;

            case 'CLEAR_OLD_CACHES':
                // Clear old caches
                clearOldCaches();
                break;

            case 'SKIP_WAITING':
                // Skip waiting and activate
                self.skipWaiting();
                break;
        }
    }
});

// Push event handler
self.addEventListener('push', event => {
    console.log('[Service Worker] Push notification received:', event);

    // Default notification data

```

```

let data = {
    title: 'CVD Risk Toolkit',
    body: 'New update available',
    icon: '/icons/icon-192x192.svg',
    data: {
        url: '/'
    }
};

// Try to parse notification data
if (event.data) {
    try {
        data = JSON.parse(event.data.text());
    } catch (error) {
        console.error('[Service Worker] Error parsing push data:', error);
    }
}

// Show notification
event.waitUntil(
    self.registration.showNotification(data.title, {
        body: data.body,
        icon: data.icon,
        badge: '/icons/badge.svg',
        data: data.data
    })
);
});

// Notification click event handler
self.addEventListener('notificationclick', event => {
    console.log('[Service Worker] Notification clicked:', event);

    // Close notification
    event.notification.close();

    // Get target URL
    let url = '/';

    if (event.notification.data && event.notification.data.url) {
        url = event.notification.data.url;
    }

    // Open URL

```

```

event.waitUntil(
  clients.matchAll({ type: 'window' })
    .then(windowClients => {
      // Find existing window
      for (const client of windowClients) {
        if (client.url === url && client.focus) {
          return client.focus();
        }
      }
    })

    // If no window found, open new one
    return clients.openWindow(url);
  })
);

// Sync event handler
self.addEventListener('sync', event => {
  console.log('[Service Worker] Background sync event:', event);

  // Handle sync events
  if (event.tag.startsWith('sync-')) {
    // Extract sync type
    const syncType = event.tag.substring(5);

    // Process sync
    event.waitUntil(
      processSyncEvent(syncType)
    );
  }
});

/***
 * Cache an asset
 * @param {string} url - URL to cache
 * @returns {Promise} - Promise that resolves when caching is complete
 */
function cacheAsset(url) {
  return caches.open(CACHE_NAMES.assets)
    .then(cache => {
      console.log('[Service Worker] Caching new asset:', url);
      return cache.add(url);
    })
    .catch(error => {

```

```

        console.error('[Service Worker] Error caching asset:', error);
    });
}

/***
 * Clear old caches
 * @returns {Promise} - Promise that resolves when clearing is complete
 */
function clearOldCaches() {
    return caches.keys()
        .then(cacheNames => {
            return Promise.all(
                cacheNames.map(cacheName => {
                    // Delete caches that don't match current version
                    if (cacheName !== CACHE_NAMES.static &&
                        cacheName !== CACHE_NAMES.dynamic &&
                        cacheName !== CACHE_NAMES.assets) {
                        console.log('[Service Worker] Removing old cache:', cacheName);
                        return caches.delete(cacheName);
                    }
                })
            );
        });
}

/***
 * Process sync event
 * @param {string} syncType - Sync type
 * @returns {Promise} - Promise that resolves when sync is complete
 */
function processSyncEvent(syncType) {
    console.log('[Service Worker] Processing sync event:', syncType);

    switch (syncType) {
        case 'data':
            // Sync data with server
            return syncData();

        case 'logs':
            // Sync Logs with server
            return syncLogs();

        default:
            console.log('[Service Worker] Unknown sync type:', syncType);
    }
}

```

```
        return Promise.resolve();
    }
}

/***
 * Sync data with server
 * @returns {Promise} - Promise that resolves when sync is complete
 */
function syncData() {
    return new Promise((resolve, reject) => {
        // Get sync queue from IndexedDB
        getFromDB('syncQueue')
            .then(queue => {
                if (!queue || queue.length === 0) {
                    // No data to sync
                    resolve();
                    return;
                }

                console.log('[Service Worker] Syncing data, items:', queue.length);

                // Process each item in queue
                return Promise.all(
                    queue.map(item => {
                        // Send data to server
                        return fetch('/api-sync', {
                            method: 'POST',
                            headers: {
                                'Content-Type': 'application/json'
                            },
                            body: JSON.stringify(item)
                        })
                            .then(response => {
                                if (response.ok) {
                                    // Remove item from queue
                                    return removeFromDB('syncQueue', item.id);
                                }
                                throw new Error('Sync failed');
                            });
                    });
            })
        .then(() => {
            console.log('[Service Worker] Data sync complete');
            resolve();
        });
    });
}
```

```
)  
    .catch(error => {  
        console.error('[Service Worker] Data sync error:', error);  
        reject(error);  
    });  
})  
    .catch(error => {  
        console.error('[Service Worker] Error getting sync queue:', error);  
        reject(error);  
    });  
});  
});  
  
/**  
 * Sync logs with server  
 * @returns {Promise} - Promise that resolves when sync is complete  
 */  
function syncLogs() {  
    return new Promise((resolve, reject) => {  
        // Get Logs from IndexedDB  
        getFromDB('logs')  
            .then(logs => {  
                if (!logs || logs.length === 0) {  
                    // No Logs to sync  
                    resolve();  
                    return;  
                }  
  
                console.log('[Service Worker] Syncing logs, items:', logs.length);  
  
                // Send Logs to server  
                return fetch('/api/logs', {  
                    method: 'POST',  
                    headers: {  
                        'Content-Type': 'application/json'  
                    },  
                    body: JSON.stringify(logs)  
                })  
            .then(response => {  
                if (response.ok) {  
                    // Clear Logs  
                    return clearDB('logs');  
                }  
                throw new Error('Log sync failed');  
            })  
        );  
    });  
}
```

```

        })
        .then(() => {
            console.log('[Service Worker] Logs sync complete');
            resolve();
        })
        .catch(error => {
            console.error('[Service Worker] Logs sync error:', error);
            reject(error);
        });
    });
}
}

/**
 * Get data from IndexedDB
 * @param {string} storeName - Store name
 * @param {string/number} id - Item ID (optional)
 * @returns {Promise} - Promise that resolves with data
 */
function getFromDB(storeName, id) {
    return new Promise((resolve, reject) => {
        // Open database
        const request = indexedDB.open('cvd-toolkit-db', 1);

        request.onerror = event => {
            reject(new Error('Failed to open database'));
        };

        request.onsuccess = event => {
            const db = event.target.result;

            // Check if store exists
            if (!db.objectStoreNames.contains(storeName)) {
                resolve(null);
                return;
            }

            // Start transaction
            const transaction = db.transaction(storeName, 'readonly');
            const store = transaction.objectStore(storeName);

```

```

let request;

if (id !== undefined) {
    // Get specific item
    request = store.get(id);
} else {
    // Get all items
    request = store.getAll();
}

request.onsuccess = event => {
    resolve(event.target.result);
};

request.onerror = event => {
    reject(new Error('Failed to get data from database'));
};

request.onupgradeneeded = event => {
    const db = event.target.result;

    // Create stores if needed
    if (!db.objectStoreNames.contains('syncQueue')) {
        db.createObjectStore('syncQueue', { keyPath: 'id' });
    }

    if (!db.objectStoreNames.contains('logs')) {
        db.createObjectStore('logs', { keyPath: 'id', autoIncrement: true });
    }
};

});

}

/***
 * Remove item from IndexedDB
 * @param {string} storeName - Store name
 * @param {string/number} id - Item ID
 * @returns {Promise} - Promise that resolves when item is removed
 */
function removeFromDB(storeName, id) {
    return new Promise((resolve, reject) => {
        // Open database

```

```

const request = indexedDB.open('cvd-toolkit-db', 1);

request.onerror = event => {
    reject(new Error('Failed to open database'));
};

request.onsuccess = event => {
    const db = event.target.result;

    // Check if store exists
    if (!db.objectStoreNames.contains(storeName)) {
        resolve();
        return;
    }

    // Start transaction
    const transaction = db.transaction(storeName, 'readwrite');
    const store = transaction.objectStore(storeName);

    // Delete item
    const request = store.delete(id);

    request.onsuccess = event => {
        resolve();
    };

    request.onerror = event => {
        reject(new Error('Failed to remove data from database'));
    };
};

});

}

/***
 * Clear IndexedDB store
 * @param {string} storeName - Store name
 * @returns {Promise} - Promise that resolves when store is cleared
 */
function clearDB(storeName) {
    return new Promise((resolve, reject) => {
        // Open database
        const request = indexedDB.open('cvd-toolkit-db', 1);

        request.onerror = event => {

```

```

        reject(new Error('Failed to open database')));
    };

    request.onerror = event => {
        const db = event.target.result;

        // Check if store exists
        if (!db.objectStoreNames.contains(storeName)) {
            resolve();
            return;
        }

        // Start transaction
        const transaction = db.transaction(storeName, 'readwrite');
        const store = transaction.objectStore(storeName);

        // Clear store
        const request = store.clear();

        request.onsuccess = event => {
            resolve();
        };

        request.onerror = event => {
            reject(new Error('Failed to clear database store'));
        };
    );
};

}

```

5.4 ui.js and styles.css Overview

These files have already been enhanced but here's a brief overview of what they contain with sample code for the new AI:

ui.js

The `ui.js` file handles all user interface interactions, DOM manipulations, and UI-related functionality. It contains functions for form validation, tooltips, modals, collapsible sections, and responsive design adjustments.

Sample code sections:

```

javascript

// Example of a UI component initialization function
function initializeTooltips() {
    // Find all elements with tooltip attribute
    const tooltipElements = document.querySelectorAll('[data-tooltip]');

    // Set up each tooltip
    tooltipElements.forEach(element => {
        // Get tooltip text
        const tooltipText = element.getAttribute('data-tooltip');

        // Set up mouse events for tooltip
        element.addEventListener('mouseenter', event => showTooltip(event, tooltipText));
        element.addEventListener('mouseleave', hideTooltip);
    });
}

// Example of a form validation function
function validateInput(input) {
    // Check browser validation
    const isValid = input.checkValidity();

    // Update input styling
    if (isValid) {
        input.classList.remove('invalid');
        input.classList.add('valid');
    } else {
        input.classList.remove('valid');
        input.classList.add('invalid');
    }

    return isValid;
}

```

The UI module handles important accessibility features, keyboard navigation, responsive behaviors, and transitions for a smooth user experience.

styles.css

The `styles.css` file contains all the styling for the CVD Risk Toolkit, including layouts, colors, typography, form elements, and responsive designs. It uses a modern, clean approach with a focus on accessibility and usability.

Sample sections:

css

```
/* Example of base styles */
:root {
    /* Color variables */
    --primary-color: #0066cc;
    --secondary-color: #ff6600;
    --text-color: #333333;
    --background-color: #ffffff;

    /* Typography variables */
    --font-family: 'Roboto', Arial, sans-serif;
    --font-size-base: 16px;
    --line-height: 1.5;

    /* Spacing variables */
    --spacing-xs: 0.25rem;
    --spacing-sm: 0.5rem;
    --spacing-md: 1rem;
    --spacing-lg: 2rem;
    --spacing-xl: 3rem;
}

/* Example of component styles */
.risk-percentage {
    font-size: 2.5rem;
    font-weight: bold;
    text-align: center;
}

.risk-percentage.low {
    color: var(--low-risk-color);
}

.risk-percentage.intermediate {
    color: var(--intermediate-risk-color);
}

.risk-percentage.high {
    color: var(--high-risk-color);
}

/* Example of responsive styles */
@media (max-width: 768px) {
    .form-container {
```

```
    flex-direction: column;
}

.form-group {
  width: 100%;
}

}
```

The CSS is organized into logical sections for:

- Base styles (variables, resets, typography)
- Layout components (grid, containers, spacing)
- Form elements (inputs, buttons, selects)
- Calculator components (risk display, results)
- Visualization elements (charts, graphs)
- Utility classes (helpers, spacing, display)
- Responsive adjustments (mobile, tablet, desktop)
- Print styles
- Animation and transitions

Conclusion

This implementation guide provides all the necessary files and code samples to create a fully functional Enhanced CVD Risk Toolkit. The remaining files to be implemented are:

1. **ui.js** - Already enhanced by you, handles all user interface interactions
2. **styles.css** - Already enhanced by you, contains all styling for the toolkit

The new AI can use the detailed sample implementations provided above to complete the toolkit. The code follows modern best practices including:

- Progressive enhancement for different device capabilities
- Robust error handling and user feedback
- Secure data handling with encryption options
- Accessibility compliance
- Offline functionality
- Cross-browser compatibility
- Mobile responsiveness

By following this guide, the new AI will be able to create a high-quality, professional application that meets all the requirements and provides an excellent user experience. // Initialize form validation
initializeFormValidation();

```
// Initialize form range inputs
initializeRangeInputs();

// Initialize form select inputs
initializeSelectInputs();

// Initialize form file inputs
initializeFileInputs();

// Initialize form auto-save
initializeFormAutoSave();

// Initialize form field dependencies
initializeFieldDependencies();
} catch (error) {
  console.error('Error initializing form UI:', error);
}

}

/**
```

- Initialize form validation */ function initializeFormValidation() { try { // Find all forms with validation const forms = document.querySelectorAll('form[data-validate]');

```

// Set up each form
forms.forEach(form => {
    // Add validation event listeners
    form.addEventListener('submit', event => {
        if (!validateForm(form)) {
            event.preventDefault();
            event.stopPropagation();

            // Show validation errors
            showValidationErrors(form);
        }
    });
});

// Add input event listeners
const inputs = form.querySelectorAll('input, select, textarea');

inputs.forEach(input => {
    // Create debounced input handler
    const debouncedHandler = RuntimeProtection.debounce(() => {
        validateInput(input);
    }, 300);

    // Add input event
    input.addEventListener('input', debouncedHandler);

    // Add blur event
    input.addEventListener('blur', () => {
        validateInput(input);
    });
});
});

} catch (error) { console.error('Error initializing form validation:', error); }

/**
```

- Validate form
- @param {HTMLFormElement} form - Form element
- @returns {boolean} - Whether form is valid */ function validateForm(form) { try { // Check browser validation if (!form.checkValidity()) { return false; }

```
// Check custom validation
const inputs = form.querySelectorAll('input, select, textarea');
let isValid = true;

inputs.forEach(input => {
  if (!validateInput(input)) {
    isValid = false;
  }
});

return isValid;

} catch (error) { console.error('Error validating form:', error); return false; }

/**
```

- Validate input
- @param {HTMLInputElement|HTMLSelectElement|HTMLTextAreaElement} input - Input element
- @returns {boolean} - Whether input is valid */ function validateInput(input) { try { // Skip disabled inputs if (input.disabled) { return true; } }

```

// Check browser validation
const isValid = input.checkValidity();

// Update input styling
if (isValid) {
    input.classList.remove('invalid');
    input.classList.add('valid');
} else {
    input.classList.remove('valid');
    input.classList.add('invalid');
}

// Find error message container
const errorContainer = input.parentNode.querySelector('.error-message');

if (errorContainer) {
    errorContainer.textContent = input.validationMessage;

    if (isValid) {
        errorContainer.style.display = 'none';
    } else {
        errorContainer.style.display = 'block';
    }
}

return isValid;
} catch (error) { console.error('Error validating input:', error); return false; }

/**
```

- Show validation errors
- @param {HTMLFormElement} form - Form element */ function showValidationErrors(form) { try { // Find first invalid input const invalidInput = form.querySelector(':invalid');

```
if (invalidInput) {  
    // Focus invalid input  
    invalidInput.focus();  
  
    // Scroll to invalid input  
    invalidInput.scrollIntoView({  
        behavior: 'smooth',  
        block: 'center'  
    });  
}  
  
} catch (error) { console.error('Error showing validation errors:', error); } }
```

```
/**
```

- Initialize range inputs */ function initializeRangeInputs() { try { // Find all range inputs const rangeInputs = document.querySelectorAll('input[type="range"]');

```
// Set up each range input
rangeInputs.forEach(range => {
    // Find or create value display
    let valueDisplay = range.parentNode.querySelector('.range-value');

    if (!valueDisplay) {
        valueDisplay = document.createElement('span');
        valueDisplay.className = 'range-value';
        range.parentNode.appendChild(valueDisplay);
    }

    // Update value display
    const updateValueDisplay = () => {
        valueDisplay.textContent = range.value;

        // Position value display
        const rangeWidth = range.offsetWidth;
        const rangeMin = parseFloat(range.min) || 0;
        const rangeMax = parseFloat(range.max) || 100;
        const rangeValue = parseFloat(range.value);

        // Calculate position percentage
        const percentage = ((rangeValue - rangeMin) / (rangeMax - rangeMin)) * 100;

        // Position value display
        valueDisplay.style.left = `${percentage}%`;
        valueDisplay.style.transform = 'translateX(-50%)';
    };

    // Update initial value
    updateValueDisplay();

    // Add input event
    range.addEventListener('input', updateValueDisplay);

    // Add change event
    range.addEventListener('change', updateValueDisplay);
});

} catch (error) { console.error('Error initializing range inputs:', error); } }

/**
```

- Initialize select inputs */ function initializeSelectInputs() { try { // Find all select inputs const selectInputs = document.querySelectorAll('select');

 // Set up each select input
 selectInputs.forEach(select => {
 // Add event listener for value changes
 select.addEventListener('change', () => {
 // Update selected option styling
 updateSelectStyling(select);
 });

 // Initialize styling
 updateSelectStyling(select);
 });

} catch (error) { console.error('Error initializing select inputs:', error); } }

/**

• Update select styling

• @param {HTMLSelectElement} select - Select element */ function updateSelectStyling(select) { try { // Check if select has a value if (select.value) { select.classList.add('has-value'); } else {
 select.classList.remove('has-value'); } } catch (error) { console.error('Error updating select styling:', error); } }

/**

• Initialize file inputs */ function initializeFileInputs() { try { // Find all file inputs const fileInputs = document.querySelectorAll('input[type="file"]');

```

// Set up each file input
fileInputs.forEach(fileInput => {
    // Find or create file name display
    let fileNameDisplay = fileInput.parentNode.querySelector('.file-name');

    if (!fileNameDisplay) {
        fileNameDisplay = document.createElement('span');
        fileNameDisplay.className = 'file-name';
        fileInput.parentNode.appendChild(fileNameDisplay);
    }

    // Handle file selection
    fileInput.addEventListener('change', () => {
        if (fileInput.files && fileInput.files.length > 0) {
            // Show file name
            fileNameDisplay.textContent = fileInput.files[0].name;
            fileNameDisplay.style.display = 'block';

            // Add has-file class
            fileInput.classList.add('has-file');
        } else {
            // Clear file name
            fileNameDisplay.textContent = '';
            fileNameDisplay.style.display = 'none';

            // Remove has-file class
            fileInput.classList.remove('has-file');
        }
    });
});

})�;
} catch (error) { console.error('Error initializing file inputs:', error); } }

/**
```

- Initialize form auto-save */ function initializeFormAutoSave() { try { // Find all forms with auto-save
const forms = document.querySelectorAll('form[data-autosave]');

```
// Set up each form
forms.forEach(form => {
    // Get form ID
    const formId = form.id || 'form';

    // Find all inputs
    const inputs = form.querySelectorAll('input, select, textarea');

    // Set up each input
    inputs.forEach(input => {
        // Skip submit buttons and file inputs
        if (input.type === 'submit' || input.type === 'file') {
            return;
        }

        // Get input ID or name
        const inputId = input.id || input.name;

        if (!inputId) {
            return;
        }

        // Create storage key
        const storageKey = `${formId}.${inputId}`;

        // Load saved value
        const savedValue = localStorage.getItem(storageKey);

        if (savedValue !== null) {
            if (input.type === 'checkbox') {
                input.checked = savedValue === 'true';
            } else if (input.type === 'radio') {
                input.checked = input.value === savedValue;
            } else {
                input.value = savedValue;
            }
        }

        // Create debounced save function
        const debouncedSave = RuntimeProtection.debounce(() => {
            // Save value
            if (input.type === 'checkbox') {
                localStorage.setItem(storageKey, input.checked);
            }
        });
    });
});
```

```

        } else if (input.type === 'radio') {
            if (input.checked) {
                localStorage.setItem(storageKey, input.value);
            }
        } else {
            localStorage.setItem(storageKey, input.value);
        }
    }, 500);

    // Add input event
    input.addEventListener('input', debouncedSave);

    // Add change event for checkboxes and radios
    if (input.type === 'checkbox' || input.type === 'radio') {
        input.addEventListener('change', debouncedSave);
    }
});

});

} catch (error) { console.error('Error initializing form auto-save:', error); }

/**
```

- Initialize field dependencies */ function initializeFieldDependencies() { try { // Find all elements with dependencies const dependentElements = document.querySelectorAll('[data-depends-on]');

```

// Set up each dependent element
dependentElements.forEach(element => {
    // Get dependency attributes
    const dependsOn = element.getAttribute('data-depends-on');
    const dependsValue = element.getAttribute('data-depends-value');
    const dependsNotValue = element.getAttribute('data-depends-not-value');
    const dependsChecked = element.hasAttribute('data-depends-checked');
    const dependsNotChecked = element.hasAttribute('data-depends-not-checked');

    // Find controlling element
    const controlElement = document.getElementById(dependsOn);

    if (!controlElement) {
        console.error(`Dependency element not found: ${dependsOn}`);
        return;
    }

    // Create update function
    const updateVisibility = () => {
        let showElement = true;

        // Check dependencies
        if (dependsChecked && controlElement.type === 'checkbox') {
            showElement = controlElement.checked;
        } else if (dependsNotChecked && controlElement.type === 'checkbox') {
            showElement = !controlElement.checked;
        } else if (dependsValue !== null) {
            showElement = controlElement.value === dependsValue;
        } else if (dependsNotValue !== null) {
            showElement = controlElement.value !== dependsNotValue;
        }

        // Update visibility
        if (showElement) {
            element.style.display = '';
        }

        // Enable form elements
        const formElements = element.querySelectorAll('input, select, textarea, button');
        formElements.forEach(el => {
            el.disabled = false;
        });
    } else {

```

```

        element.style.display = 'none';

        // Disable form elements
        const formElements = element.querySelectorAll('input, select, textarea,
button');
        formElements.forEach(el => {
            el.disabled = true;
        });
    }
};

// Run initial update
updateVisibility();

// Add event listeners
controlElement.addEventListener('change', updateVisibility);
controlElement.addEventListener('input', updateVisibility);
});

} catch (error) { console.error('Error initializing field dependencies:', error); } }

/**/

```

- Initialize responsive behaviors */ function initializeResponsiveBehaviors() { try { // Initialize responsive navigation
initializeResponsiveNavigation();

```

        // Initialize responsive tables
initializeResponsiveTables();

        // Set up resize event handler
window.addEventListener('resize', RuntimeProtection.debounce(() => {
        // Update responsive elements
        updateResponsiveElements();
    }, 200));

        // Run initial update
updateResponsiveElements();
}

} catch (error) { console.error('Error initializing responsive behaviors:', error); } }

/**/

```

- Initialize responsive navigation */ function initializeResponsiveNavigation() { try { // Find navigation toggle button
const navToggle = document.querySelector('.nav-toggle');

```

if (!navToggle) {
    return;
}

// Find navigation container
const navContainer = document.querySelector('.nav-container');

if (!navContainer) {
    return;
}

// Add click event
navToggle.addEventListener('click', () => {
    // Toggle navigation
    navContainer.classList.toggle('open');

    // Update button state
    const isOpen = navContainer.classList.contains('open');
    navToggle.setAttribute('aria-expanded', isOpen.toString());
});

// Close navigation when clicking outside
document.addEventListener('click', event => {
    if (navContainer.classList.contains('open') &&
        !navContainer.contains(event.target) &&
        !navToggle.contains(event.target)) {

        // Close navigation
        navContainer.classList.remove('open');
        navToggle.setAttribute('aria-expanded', 'false');
    }
});
}

} catch (error) { console.error('Error initializing responsive navigation:', error); } }

/**
```

- Initialize responsive tables */ function initializeResponsiveTables() { try { // Find all tables const tables = document.querySelectorAll('table');

```

// Set up each table
tables.forEach(table => {
    // Skip tables with no-responsive class
    if (table.classList.contains('no-responsive')) {
        return;
    }

    // Add responsive class
    table.classList.add('responsive-table');

    // Find all table headers
    const headers = Array.from(table.querySelectorAll('th')).map(th =>
        th.textContent.trim());

    // Find all data cells
    const dataCells = table.querySelectorAll('tbody td');

    // Add data attributes to cells
    dataCells.forEach((cell, index) => {
        const headerIndex = index % headers.length;
        cell.setAttribute('data-label', headers[headerIndex]);
    });
});

} catch (error) { console.error('Error initializing responsive tables:', error); }

/**

```

- Update responsive elements */
function updateResponsiveElements() {
try {
// Get window width
const windowHeight = window.innerWidth;

// Update body class based on screen size
if (windowWidth < 768) {
 document.body.classList.add('small-screen');
 document.body.classList.remove('medium-screen', 'large-screen');
} else if (windowWidth < 1024) {
 document.body.classList.add('medium-screen');
 document.body.classList.remove('small-screen', 'large-screen');
} else {
 document.body.classList.add('large-screen');
 document.body.classList.remove('small-screen', 'medium-screen');
}

} catch (error) { console.error('Error updating responsive elements:', error); }
}

/**

- Initialize theme switching */ function initializeThemeSwitching() { try { // Find theme switch element const themeSwitch = document.querySelector('.theme-switch');

```
if (!themeSwitch) {
    return;
}

// Check for saved theme
const savedTheme = localStorage.getItem('theme');

if (savedTheme) {
    // Apply saved theme
    document.documentElement.setAttribute('data-theme', savedTheme);

    // Update switch state
    if (themeSwitch.type === 'checkbox') {
        themeSwitch.checked = savedTheme === 'dark';
    }
} else if (window.matchMedia && window.matchMedia('(prefers-color-scheme: dark)').matches) {
    // Use dark theme if user prefers it
    document.documentElement.setAttribute('data-theme', 'dark');

    // Update switch state
    if (themeSwitch.type === 'checkbox') {
        themeSwitch.checked = true;
    }
}

// Add change event
themeSwitch.addEventListener('change', () => {
    if (themeSwitch.type === 'checkbox') {
        // Toggle theme based on checkbox
        const theme = themeSwitch.checked ? 'dark' : 'light';
        document.documentElement.setAttribute('data-theme', theme);
        localStorage.setItem('theme', theme);
    } else {
        // Toggle between light and dark
        const currentTheme = document.documentElement.getAttribute('data-theme') ||
'light';
        const newTheme = currentTheme === 'dark' ? 'light' : 'dark';
        document.documentElement.setAttribute('data-theme', newTheme);
        localStorage.setItem('theme', newTheme);
    }
});
```

```
    } catch (error) { console.error('Error initializing theme switching:', error); } }
```

```
/**
```

- Initialize printing */ function initializePrinting() { try { // Find print buttons const printButtons = document.querySelectorAll('.print-button');

```
    // Set up each print button
    printButtons.forEach(button => {
        button.addEventListener('click', () => {
            // Get target ID
            const targetId = button.getAttribute('data-print-target');

            if (targetId) {
                // Print specific element
                printElement(targetId);
            } else {
                // Print entire page
                window.print();
            }
        });
    });
}
```

```
} catch (error) { console.error('Error initializing printing:', error); } }
```

```
/**
```

- Print specific element
- @param {string} elementId - Element ID to print */ function printElement(elementId) { try { // Find element const element = document.getElementById(elementId);

```
if (!element) {
    console.error(`Print element not found: ${elementId}`);
    return;
}

// Create iframe
const iframe = document.createElement('iframe');
iframe.style.display = 'none';
document.body.appendChild(iframe);

// Copy styles
const styles = Array.from(document.styleSheets).map(styleSheet => {
    try {
        if (styleSheet.cssRules) {
            return Array.from(styleSheet.cssRules)
                .map(rule => rule.cssText)
                .join('\n');
        }
        return '';
    } catch (e) {
        console.warn('Error accessing stylesheet:', e);
        return '';
    }
}).join('\n');

// Write content to iframe
const iframeDocument = iframe.contentDocument || iframe.contentWindow.document;
iframeDocument.open();
iframeDocument.write(`
<!DOCTYPE html>
<html>
<head>
    <title>Print</title>
    <style>
        ${styles}
        body {
            font-family: Arial, sans-serif;
            line-height: 1.5;
            color: #333;
            background: white;
        }
        @media print {
            body {

```

```

        padding: 0;
        margin: 0;
    }
}
</style>
</head>
<body>
${element.outerHTML}
</body>
</html>
`);

iframeDocument.close();

// Wait for styles to load
setTimeout(() => {
    // Print iframe
    iframe.contentWindow.focus();
    iframe.contentWindow.print();

    // Remove iframe after printing
    setTimeout(() => {
        document.body.removeChild(iframe);
    }, 1000);
}, 500);

} catch (error) { console.error('Error printing element:', error); }

/***
• Initialize keyboard navigation */
function initializeKeyboardNavigation() {
try {
    // Add keyboard event
    listener
    document.addEventListener('keydown', event => {
        // Skip if in input field if
        (event.target.tagName === 'INPUT' || event.target.tagName === 'TEXTAREA' || event.target.tagName === 'SELECT' || event.target.isContentEditable) { return; }
    })
}

```

```
// Handle keyboard shortcuts
if (event.ctrlKey || event.metaKey) {
    // Ctrl/Cmd + S: Save
    if (event.key === 's') {
        event.preventDefault();

        // Trigger save button
        const saveButton = document.querySelector('.save-button');

        if (saveButton) {
            saveButton.click();
        }
    }

    // Ctrl/Cmd + P: Print
    if (event.key === 'p') {
        // Browser will handle print
        // No need to prevent default
    }
} else {
    // Navigation keys
    switch (event.key) {
        case 'h':
            // Navigate to home
            navigateToTab('home');
            break;

        case 'f':
            // Navigate to FRS calculator
            navigateToTab('frs');
            break;

        case 'q':
            // Navigate to QRISK calculator
            navigateToTab('qrisk');
            break;

        case 'c':
            // Navigate to combined view
            navigateToTab('combined');
            break;

        case 'm':
```

```

        // Navigate to medication
        navigateToTab('medication');
        break;

    case 'v':
        // Navigate to visualization
        navigateToTab('visualization');
        break;

    case '?':
        // Show keyboard shortcuts
        showKeyboardShortcuts();
        break;
    }
}
});

} catch (error) { console.error('Error initializing keyboard navigation:', error); } }

/**

```

- Navigate to tab
- @param {string} tabId - Tab ID */ function navigateToTab(tabId) { try { // Find tab const tab = document.querySelector([data-tab="\${tabId}"]);

```

    if (tab) {
        tab.click();
    }
}

} catch (error) { console.error('Error navigating to tab:', error); } }

/**

```

- Show keyboard shortcuts */ function showKeyboardShortcuts() { try { // Find or create shortcut modal let shortcutModal = document.getElementById('keyboard-shortcuts-modal');

```
if (!shortcutModal) {  
    shortcutModal = document.createElement('div');  
    shortcutModal.id = 'keyboard-shortcuts-modal';  
    shortcutModal.className = 'modal';  
    shortcutModal.innerHTML = `  
        <div class="modal-header">  
            <h3>Keyboard Shortcuts</h3>  
            <button type="button" class="modal-close">&times;</button>  
        </div>  
        <div class="modal-body">  
            <table class="shortcuts-table">  
                <thead>  
                    <tr>  
                        <th>Key</th>  
                        <th>Action</th>  
                    </tr>  
                </thead>  
                <tbody>  
                    <tr>  
                        <td>h</td>  
                        <td>Go to Home</td>  
                    </tr>  
                    <tr>  
                        <td>f</td>  
                        <td>Go to Framingham Calculator</td>  
                    </tr>  
                    <tr>  
                        <td>q</td>  
                        <td>Go to QRISK Calculator</td>  
                    </tr>  
                    <tr>  
                        <td>c</td>  
                        <td>Go to Combined View</td>  
                    </tr>  
                    <tr>  
                        <td>m</td>  
                        <td>Go to Medication</td>  
                    </tr>  
                    <tr>  
                        <td>v</td>  
                        <td>Go to Visualization</td>  
                    </tr>  
                    <tr>
```

```

        <td>Ctrl/Cmd + S</td>
        <td>Save</td>
    </tr>
    <tr>
        <td>Ctrl/Cmd + P</td>
        <td>Print</td>
    </tr>
    <tr>
        <td>?</td>
        <td>Show Keyboard Shortcuts</td>
    </tr>
</tbody>
</table>
</div>
`;
document.body.appendChild(shortcutModal);

// Add close button event
const closeButton = shortcutModal.querySelector('.modal-close');
if (closeButton) {
    closeButton.addEventListener('click', () => {
        hideModal('keyboard-shortcuts-modal');
    });
}

// Show modal
showModal('keyboard-shortcuts-modal');

} catch (error) { console.error('Error showing keyboard shortcuts:', error); } }

/**

```

- Initialize clipboard functionality */ function initializeClipboard() { try { // Find clipboard buttons const clipboardButtons = document.querySelectorAll('.clipboard-button');

```

// Set up each button
clipboardButtons.forEach(button => {
    button.addEventListener('click', () => {
        // Get target ID
        const targetId = button.getAttribute('data-clipboard-target');

        if (!targetId) {
            return;
        }

        // Find target element
        const targetElement = document.getElementById(targetId);

        if (!targetElement) {
            console.error(`Clipboard target not found: ${targetId}`);
            return;
        }

        // Copy text to clipboard
        copyToClipboard(targetElement.innerText || targetElement.textContent);

        // Show success message
        const successMessage = button.getAttribute('data-clipboard-success') ||
'Copied!';

        // Update button text temporarily
        const originalText = button.innerText;
        button.innerText = successMessage;

        // Restore original text after delay
        setTimeout(() => {
            button.innerText = originalText;
        }, 2000);
    });
});

} catch (error) { console.error('Error initializing clipboard:', error); }

/**
```

- Copy text to clipboard
- @param {string} text - Text to copy

- @returns {boolean} - Whether copy was successful */ function copyToClipboard(text) { try { // Use Clipboard API if available if (navigator.clipboard && window.isSecureContext) { navigator.clipboard.writeText(text); return true; }

 // Fallback method
 const textarea = document.createElement('textarea');
 textarea.value = text;
 textarea.style.position = 'fixed';
 textarea.style.opacity = '0';
 document.body.appendChild(textarea);
 textarea.select();

 // Copy text
 const success = document.execCommand('copy');

 // Clean up
 document.body.removeChild(textarea);

 return success;
 } catch (error) { console.error('Error copying to clipboard:', error); return false; } }

// Export for module usage export default { initialize: initializeUI };

```
### 5.3 service-worker.js

```javascript
/**
 * Service Worker for CVD Risk Toolkit
 *
 * Provides offline functionality, caching, and
 * background synchronization features.
 */

// Service worker version
const VERSION = '1.0.0';

// Cache names
const CACHE_NAMES = {
 static: `static-cache-${VERSION}`,
 dynamic: `dynamic-cache-${VERSION}`,
 assets: `assets-cache-${VERSION}`
};

// Resources to cache on install
const STATIC_RESOURCES = [
 '/',
 '/index.html',
 '/css/styles.css',
 '/js/main.js',
 '/js/ui.js',
 '/offline.html'
];

// Install event handler
self.addEventListener('install', event => {
 console.log('[Service Worker] Installing service worker, version:', VERSION);

 // Skip waiting to activate immediately
 self.skipWaiting();

 // Cache static resources
 event.waitUntil(
 caches.open(CACHE_NAMES.static)
 .then(cache => {
 console.log('[Service Worker] Caching static resources');
 })
);
});
```

```

 return cache.addAll(STATIC_RESOURCES);
 })
 .catch(error => {
 console.error('[Service Worker] Error caching static resources:', error);
 })
);

// Activate event handler
self.addEventListener('activate', event => {
 console.log('[Service Worker] Activating service worker, version:', VERSION);

 // Clean up old caches
 event.waitUntil(
 caches.keys()
 .then(cacheNames => {
 return Promise.all(
 cacheNames.map(cacheName => {
 // Delete caches that don't match current version
 if (cacheName !== CACHE_NAMES.static &&
 cacheName !== CACHE_NAMES.dynamic &&
 cacheName !== CACHE_NAMES.assets) {
 console.log('[Service Worker] Removing old cache:', cacheName);
 return caches.delete(cacheName);
 }
 })
);
 })
 .then(() => {
 // Claim clients to take control immediately
 return self.clients.claim();
 })
);
});

// Fetch event handler
self.addEventListener('fetch', event => {
 // Skip cross-origin requests
 initializeVisualizations(ChartRenderer) {
 try {
 // Get visualization form
 const visualizationForm = document.getElementById('visualization-form');

 if (!visualizationForm) return;

```

```
// Get chart container
const chartContainer = document.getElementById('visualization-chart');

if (!chartContainer) return;

// Set up form submission
visualizationForm.addEventListener('submit', async (event) => {
 event.preventDefault();

 // Show loading indicator
 LoadingManager.showLoading('Generating visualization...');

 // Get form data
 const formData = new FormData(visualizationForm);
 const data = {};

 // Convert form data to object
 for (const [key, value] of formData.entries()) {
 data[key] = value;
 }

 try {
 // Render chart
 const result = await ChartRenderer.renderChart(chartContainer, data);

 // Show details if available
 const detailsContainer = document.getElementById('visualization-details');

 if (detailsContainer && result.details) {
 detailsContainer.innerHTML = result.details;
 detailsContainer.style.display = 'block';
 }

 // Enable export button
 const exportButton = document.getElementById('export-visualization-btn');

 if (exportButton) {
 exportButton.disabled = false;
 }
 } catch (error) {
 console.error('Error rendering chart:', error);

 // Show error notification
 ErrorDetectionSystem.showErrorNotification(

```

```

 new Error(`Error generating visualization: ${error.message}`),
 'Visualization Error'
);
}

// Hide loading indicator
LoadingManager.hideLoading();
});

// Set up export button
const exportButton = document.getElementById('export-visualization-btn');

if (exportButton) {
 exportButton.addEventListener('click', async () => {
 try {
 // Import chart exporter
 const ChartExporter = (await
ModuleLoader.loadModule('./visualizations/chart-exporter.js')).default;

 // Get export format
 const format = exportButton.getAttribute('data-format') || 'pdf';

 // Export chart
 const result = await ChartExporter.exportChart(chartContainer, format);

 if (!result) {
 throw new Error('Export failed');
 }
 } catch (error) {
 console.error('Error exporting chart:', error);

 // Show error notification
 ErrorDetectionSystem.showErrorNotification(
 new Error(`Error exporting chart: ${error.message}`),
 'Export Error'
);
 }
 });
}

// Disable button initially
exportButton.disabled = true;
}

} catch (error) {
 console.error('Error initializing visualizations:', error);
}

```

```
 }

}

/***
 * Initialize offline detection
 */
function initializeOfflineDetection() {
 try {
 // Check initial status
 updateOnlineStatus();

 // Set up event listeners for online/offline status
 window.addEventListener('online', updateOnlineStatus);
 window.addEventListener('offline', updateOnlineStatus);
 } catch (error) {
 console.error('Error initializing offline detection:', error);
 }
}

/***
 * Update online status
 */
function updateOnlineStatus() {
 try {
 if (navigator.onLine) {
 // Online
 document.body.classList.remove('offline');
 document.body.classList.add('online');

 // Show notification if transitioning from offline
 if (document.body.hasAttribute('data-was-offline')) {
 document.body.removeAttribute('data-was-offline');
 ErrorDetectionSystem.showOnlineNotification();
 }
 } else {
 // Offline
 document.body.classList.remove('online');
 document.body.classList.add('offline');
 document.body.setAttribute('data-was-offline', 'true');

 // Show notification
 ErrorDetectionSystem.showOfflineNotification();
 }
 } catch (error) {
}
```

```

 console.error('Error updating online status:', error);
 }
}

/***
 * Show help for specific topic
 * @param {string} helpId - Help topic ID
 */
function showHelp(helpId) {
 try {
 // Import help system
 import('./utils/help-system.js').then(module => {
 const HelpSystem = module.default;
 HelpSystem.showHelp(helpId);
 }).catch(error => {
 console.error('Error loading help system:', error);
 });
 } catch (error) {
 console.error('Error showing help:', error);
 }
}

/***
 * Export data in specified format
 * @param {string} exportId - Export element ID
 * @param {string} format - Export format
 */
function exportData(exportId, format) {
 try {
 // Import data exporter
 import('./data-management/data-import-export.js').then(module => {
 const DataImportExport = module.default;

 // Get data to export
 let data = null;

 if (exportId === 'frs-results') {
 const frsResultJson = sessionStorage.getItem('frs_result');
 if (frsResultJson) {
 data = JSON.parse(frsResultJson);
 }
 } else if (exportId === 'qrisk-results') {
 const qriskResultJson = sessionStorage.getItem('qrisk_result');
 if (qriskResultJson) {

```

```

 data = JSON.parse(qriskResultJson);
 }
} else if (exportId === 'medication-recommendations') {
 const recommendationsJson = sessionStorage.getItem('recommendations');
 if (recommendationsJson) {
 data = JSON.parse(recommendationsJson);
 }
}

if (!data) {
 ErrorDetectionSystem.showErrorNotification(
 new Error('No data available to export'),
 'Export Error'
);
 return;
}

// Export data
DataImportExport.exportData(data, format, false, true)
 .then(result => {
 if (!result.success) {
 throw new Error(result.error || 'Export failed');
 }
 })
 .catch(error => {
 console.error('Error exporting data:', error);

 // Show error notification
 ErrorDetectionSystem.showErrorNotification(
 new Error(`Error exporting data: ${error.message}`),
 'Export Error'
);
 });
}).catch(error => {
 console.error('Error loading data exporter:', error);
});

} catch (error) {
 console.error('Error exporting data:', error);
}
}

/***
 * Process form submission
 * @param {HTMLFormElement} form - Form element

```

```

*/
function processForm(form) {
 try {
 // Get form ID
 const formId = form.id;

 // Process based on form ID
 if (formId === 'frs-form' || formId === 'qrisk-form') {
 // Get calculator type
 const calculatorType = formId.split('-')[0];

 // Calculate risk
 calculateRisk(form, calculatorType);
 } else if (formId === 'medication-form') {
 // Submit directly (event handlers are already set up)
 // This will trigger the submit event again, but the event.preventDefault()
 prevents infinite loop
 } else if (formId === 'visualization-form') {
 // Submit directly (event handlers are already set up)
 } else if (formId === 'import-form') {
 // Import data
 importData(form);
 } else if (formId === 'export-form') {
 // Export data
 exportDataFromForm(form);
 }
 } catch (error) {
 console.error('Error processing form:', error);
 }
}

/**
 * Import data from file
 * @param {HTMLFormElement} form - Import form
 */
async function importData(form) {
 try {
 // Show loading indicator
 LoadingManager.showLoading('Importing data...');

 // Get file input
 const fileInput = form.querySelector('input[type="file"]');

 if (!fileInput || !fileInput.files || fileInput.files.length === 0) {

```

```
 throw new Error('No file selected');

 }

// Get selected file
const file = fileInput.files[0];

// Get format
const formatSelect = form.querySelector('select[name="format"]');
const format = formatSelect ? formatSelect.value : 'json';

// Get decrypt option
const decryptCheckbox = form.querySelector('input[name="decrypt"]');
const decrypt = decryptCheckbox ? decryptCheckbox.checked : false;

// Import data
const DataImportExport = (await ModuleLoader.loadModule('./data-management/data-import-export.js')).default;

const result = await DataImportExport.importData(file, format, decrypt);

if (!result.success) {
 throw new Error(result.error || 'Import failed');
}

// Load imported data
await DataImportExport.loadImportedData(result.data);

// Show success notification
ErrorDetectionSystem.showSuccessNotification('Data imported successfully');

// Clear file input
fileInput.value = '';

// Hide loading indicator
LoadingManager.hideLoading();
} catch (error) {
 console.error('Error importing data:', error);

 // Show error notification
 ErrorDetectionSystem.showErrorNotification(
 new Error(`Error importing data: ${error.message}`),
 'Import Error'
);
}
```

```
// Hide loading indicator
LoadingManager.hideLoading();
}

}

/***
 * Export data from export form
 * @param {HTMLFormElement} form - Export form
 */
async function exportDataFromForm(form) {
 try {
 // Show loading indicator
 LoadingManager.showLoading('Exporting data...');

 // Get format
 const formatSelect = form.querySelector('select[name="format"]');
 const format = formatSelect ? formatSelect.value : 'json';

 // Get encrypt option
 const encryptCheckbox = form.querySelector('input[name="encrypt"]');
 const encrypt = encryptCheckbox ? encryptCheckbox.checked : false;

 // Get include recommendations option
 const includeRecommendationsCheckbox = form.querySelector('input[name="include-recommendations"]');
 const includeRecommendations = includeRecommendationsCheckbox ?
 includeRecommendationsCheckbox.checked : true;

 // Get data to export
 const data = {};

 // Add patient data
 const patientData = await getPatientData();

 if (patientData) {
 data.patientData = patientData;
 }

 // Add saved results
 try {
 const frsResultJson = sessionStorage.getItem('frs_result');
 if (frsResultJson) {
 data.frsResult = JSON.parse(frsResultJson);
 }
 }
 }
}
```

```

 const qriskResultJson = sessionStorage.getItem('qrisk_result');
 if (qriskResultJson) {
 data.qriskResult = JSON.parse(qriskResultJson);
 }

 // Add recommendations if requested
 if (includeRecommendations) {
 const recommendationsJson = sessionStorage.getItem('recommendations');
 if (recommendationsJson) {
 data.recommendations = JSON.parse(recommendationsJson);
 }
 }
 } catch (e) {
 console.error('Error retrieving saved results:', e);
 }

 // Check if data is empty
 if (Object.keys(data).length === 0) {
 throw new Error('No data available to export');
 }

 // Export data
 const DataImportExport = (await ModuleLoader.loadModule('./data-management/data-import-export.js')).default;

 const result = await DataImportExport.exportData(data, format, encrypt,
includeRecommendations);

 if (!result.success) {
 throw new Error(result.error || 'Export failed');
 }

 // Show success notification
 ErrorDetectionSystem.showSuccessNotification('Data exported successfully');

 // Hide loading indicator
 LoadingManager.hideLoading();
} catch (error) {
 console.error('Error exporting data:', error);

 // Show error notification
 ErrorDetectionSystem.showErrorNotification(
 new Error(`Error exporting data: ${error.message}`),

```

```
'Export Error'

);

// Hide loading indicator
LoadingManager.hideLoading();
}

}

/***
 * Get patient data from forms
 * @returns {Promise<Object>} - Patient data
 */
async function getPatientData() {
 try {
 // Try to get patient data from medication form
 const medicationForm = document.getElementById('medication-form');

 if (medicationForm) {
 const formData = new FormData(medicationForm);
 const data = {};

 // Convert form data to object
 for (const [key, value] of formData.entries()) {
 data[key] = value;
 }

 // Add checkbox values
 const checkboxes = medicationForm.querySelectorAll('input[type="checkbox"]');
 checkboxes.forEach(checkbox => {
 data[checkbox.name] = checkbox.checked;
 });
 }

 return data;
 }

 // Try to get patient data from FRS form
 const frsForm = document.getElementById('frs-form');

 if (frsForm) {
 const formData = new FormData(frsForm);
 const data = {};

 // Convert form data to object
 for (const [key, value] of formData.entries()) {
```

```
 data[key] = value;
 }

 // Add checkbox values
 const checkboxes = frsForm.querySelectorAll('input[type="checkbox"]');
 checkboxes.forEach(checkbox => {
 data[checkbox.name] = checkbox.checked;
 });

 return data;
}

// Try to get patient data from QRISK form
const qriskForm = document.getElementById('qrisk-form');

if (qriskForm) {
 const formData = new FormData(qriskForm);
 const data = {};

 // Convert form data to object
 for (const [key, value] of formData.entries()) {
 data[key] = value;
 }

 // Add checkbox values
 const checkboxes = qriskForm.querySelectorAll('input[type="checkbox"]');
 checkboxes.forEach(checkbox => {
 data[checkbox.name] = checkbox.checked;
 });

 return data;
}

return null;
} catch (error) {
 console.error('Error getting patient data:', error);
 return null;
}
}

// Export for module usage
export default {
```

```
 initialize: initializeApp
};
```

## 5.2 ui.js

javascript

```
/**
 * UI Interactions for CVD Risk Toolkit
 *
 * Manages UI interactions, animations, tooltips, modals,
 * and other UI-related functionality.
 */

import RuntimeProtection from './utils/runtime-protection.js';

// Initialize UI when DOM is Loaded
document.addEventListener('DOMContentLoaded', initializeUI);

/**
 * Initialize UI
 */
function initializeUI() {
 try {
 console.log('Initializing UI...');

 // Initialize tooltips
 initializeTooltips();

 // Initialize modals
 initializeModals();

 // Initialize collapsible sections
 initializeCollapsibles();

 // Initialize dropdowns
 initializeDropdowns();

 // Initialize form UI enhancements
 initializeFormUI();

 // Initialize responsive behaviors
 initializeResponsiveBehaviors();

 // Initialize theme switching
 initializeThemeSwitching();

 // Initialize print functionality
 initializePrinting();
 } catch (error) {
 console.error('Error initializing UI:', error);
 }
}
// Initialize UI when DOM is Loaded
document.addEventListener('DOMContentLoaded', initializeUI);
```

```
// Initialize keyboard navigation
initializeKeyboardNavigation();

// Initialize clipboard functionality
initializeClipboard();

 console.log('UI initialization complete');
} catch (error) {
 console.error('Error initializing UI:', error);
}

}

/***
 * Initialize tooltips
*/
function initializeTooltips() {
 try {
 // Find all elements with tooltip attribute
 const tooltipElements = document.querySelectorAll('[data-tooltip]');

 // Set up each tooltip
 tooltipElements.forEach(element => {
 // Get tooltip text
 const tooltipText = element.getAttribute('data-tooltip');

 // Set up mouse events for tooltip
 element.addEventListener('mouseenter', event => showTooltip(event, tooltipText));
 element.addEventListener('mouseleave', hideTooltip);
 element.addEventListener('focus', event => showTooltip(event, tooltipText));
 element.addEventListener('blur', hideTooltip);

 // Add aria attributes
 element.setAttribute('aria-describedby', `tooltip-${Math.random().toString(36).substr(2)}`);
 });
 } catch (error) {
 console.error('Error initializing tooltips:', error);
 }
}

/***
 * Show tooltip
 * @param {Event} event - Mouse event
 * @param {string} text - Tooltip text
*/

```

```
function showTooltip(event, text) {
 try {
 // Hide any existing tooltips
 hideTooltip();

 // Create tooltip element
 const tooltip = document.createElement('div');
 tooltip.id = event.target.getAttribute('aria-describedby');
 tooltip.className = 'tooltip';
 tooltip.textContent = text;

 // Position tooltip
 const rect = event.target.getBoundingClientRect();
 tooltip.style.top = `${rect.bottom + window.scrollY + 5}px`;
 tooltip.style.left = `${rect.left + window.scrollX + (rect.width / 2)}px`;
 tooltip.style.transform = 'translateX(-50%)';

 // Add tooltip to document
 document.body.appendChild(tooltip);

 // Fade in tooltip
 setTimeout(() => {
 tooltip.classList.add('visible');
 }, 10);
 } catch (error) {
 console.error('Error showing tooltip:', error);
 }
}

/**
 * Hide tooltip
 */
function hideTooltip() {
 try {
 // Find all visible tooltips
 const tooltips = document.querySelectorAll('.tooltip');

 // Remove each tooltip
 tooltips.forEach(tooltip => {
 // Fade out
 tooltip.classList.remove('visible');

 // Remove after animation
 setTimeout(() => {

```

```
 if (tooltip && tooltip.parentNode) {
 tooltip.parentNode.removeChild(tooltip);
 }
 }, 200);
});

} catch (error) {
 console.error('Error hiding tooltip:', error);
}

}

/***
 * Initialize modals
*/
function initializeModals() {
 try {
 // Find all modal triggers
 const modalTriggers = document.querySelectorAll('[data-modal]');

 // Set up each modal trigger
 modalTriggers.forEach(trigger => {
 // Get modal ID
 const modalId = trigger.getAttribute('data-modal');

 // Add click event
 trigger.addEventListener('click', () => {
 showModal(modalId);
 });
 });

 // Set up close buttons
 const closeButton = document.querySelectorAll('.modal-close');

 closeButton.forEach(button => {
 button.addEventListener('click', event => {
 // Find parent modal
 const modal = event.target.closest('.modal');

 if (modal) {
 hideModal(modal.id);
 }
 });
 });

 // Set up overlay click to close
 }
}
```

```

document.addEventListener('click', event => {
 if (event.target.classList.contains('modal-overlay')) {
 // Find modal inside overlay
 const modal = event.target.querySelector('.modal');

 if (modal) {
 hideModal(modal.id);
 }
 }
});

// Set up escape key to close modal
document.addEventListener('keydown', event => {
 if (event.key === 'Escape') {
 // Find visible modal
 const visibleModal = document.querySelector('.modal-overlay.visible');

 if (visibleModal) {
 const modal = visibleModal.querySelector('.modal');

 if (modal) {
 hideModal(modal.id);
 }
 }
 }
});

} catch (error) {
 console.error('Error initializing modals:', error);
}

}

/***
 * Show modal
 * @param {string} modalId - Modal element ID
 */
function showModal(modalId) {
 try {
 // Find modal
 const modal = document.getElementById(modalId);

 if (!modal) {
 console.error(`Modal not found: ${modalId}`);
 return;
 }
 }
}

```

```
// Find or create overlay
let overlay = document.querySelector('.modal-overlay');

if (!overlay) {
 overlay = document.createElement('div');
 overlay.className = 'modal-overlay';
 document.body.appendChild(overlay);
}

// Move modal to overlay
overlay.appendChild(modal);

// Show modal and overlay
modal.classList.add('visible');
overlay.classList.add('visible');

// Prevent body scrolling
document.body.classList.add('modal-open');

// Focus first focusable element
const focusableElements = modal.querySelectorAll('button, [href], input, select, textarea');

if (focusableElements.length > 0) {
 focusableElements[0].focus();
}

} catch (error) {
 console.error('Error showing modal:', error);
}

}

/***
 * Hide modal
 * @param {string} modalId - Modal element ID
 */
function hideModal(modalId) {
 try {
 // Find modal
 const modal = document.getElementById(modalId);

 if (!modal) {
 console.error(`Modal not found: ${modalId}`);
 return;
 }
 }
}
```

```
// Find overlay
const overlay = document.querySelector('.modal-overlay');

if (!overlay) {
 return;
}

// Hide modal and overlay
modal.classList.remove('visible');
overlay.classList.remove('visible');

// Move modal back to original location
const originalLocation = document.querySelector(`[data-modal="${modalId}"]`);

if (originalLocation) {
 const container = originalLocation.parentNode;

 if (container) {
 container.appendChild(modal);
 }
}

// Allow body scrolling
document.body.classList.remove('modal-open');

// Remove overlay after animation
setTimeout(() => {
 if (overlay.parentNode) {
 overlay.parentNode.removeChild(overlay);
 }
}, 300);
} catch (error) {
 console.error('Error hiding modal:', error);
}

/***
 * Initialize collapsible sections
 */
function initializeCollapsibles() {
 try {
 // Find all collapsible triggers
 const collapsibleTriggers = document.querySelectorAll('[data-collapse]');
 }
}
```

```
// Set up each trigger
collapsibleTriggers.forEach(trigger => {
 // Get target ID
 const targetId = trigger.getAttribute('data-collapse');

 // Find target
 const target = document.getElementById(targetId);

 if (!target) {
 console.error(`Collapse target not found: ${targetId}`);
 return;
 }

 // Set initial state
 if (trigger.hasAttribute('data-collapse-open')) {
 target.style.display = 'block';
 trigger.setAttribute('aria-expanded', 'true');
 } else {
 target.style.display = 'none';
 trigger.setAttribute('aria-expanded', 'false');
 }

 // Add click event
 trigger.addEventListener('click', () => {
 toggleCollapsible(trigger, target);
 });

 // Add accessibility attributes
 trigger.setAttribute('role', 'button');
 trigger.setAttribute('aria-controls', targetId);

 if (!trigger.hasAttribute('tabindex')) {
 trigger.setAttribute('tabindex', '0');
 }

 // Add keyboard event
 trigger.addEventListener('keydown', event => {
 if (event.key === 'Enter' || event.key === ' ') {
 event.preventDefault();
 toggleCollapsible(trigger, target);
 }
 });
});
```

```
 } catch (error) {
 console.error('Error initializing collapsibles:', error);
 }
 }

/**
 * Toggle collapsible section
 * @param {Element} trigger - Trigger element
 * @param {Element} target - Target element
 */
function toggleCollapsible(trigger, target) {
 try {
 // Check current state
 const isExpanded = trigger.getAttribute('aria-expanded') === 'true';

 if (isExpanded) {
 // Collapse
 target.style.height = `${target.scrollHeight}px`;

 // Trigger reflow
 target.offsetHeight;

 // Start transition
 target.style.height = '0';

 // Update attributes
 trigger.setAttribute('aria-expanded', 'false');

 // Hide after transition
 setTimeout(() => {
 target.style.display = 'none';
 target.style.height = '';
 }, 300);
 } else {
 // Expand
 target.style.display = 'block';
 target.style.height = '0';

 // Trigger reflow
 target.offsetHeight;

 // Start transition
 target.style.height = `${target.scrollHeight}px`;
 }
 } catch (error) {
 console.error('Error initializing collapsibles:', error);
 }
}
```

```
// Update attributes
trigger.setAttribute('aria-expanded', 'true');

// Reset height after transition
setTimeout(() => {
 target.style.height = '';
}, 300);
}

} catch (error) {
 console.error('Error toggling collapsible:', error);
}

}

/***
 * Initialize dropdowns
*/
function initializeDropdowns() {
 try {
 // Find all dropdown triggers
 const dropdownTriggers = document.querySelectorAll('[data-dropdown]');

 // Set up each trigger
 dropdownTriggers.forEach(trigger => {
 // Get target ID
 const targetId = trigger.getAttribute('data-dropdown');

 // Find target
 const target = document.getElementById(targetId);

 if (!target) {
 console.error(`Dropdown target not found: ${targetId}`);
 return;
 }

 // Set initial state
 target.style.display = 'none';
 trigger.setAttribute('aria-expanded', 'false');

 // Add click event
 trigger.addEventListener('click', event => {
 event.stopPropagation();
 toggleDropdown(trigger, target);
 });
 });
 }
}
```

```

 // Add accessibility attributes
 trigger.setAttribute('aria-controls', targetId);

 if (!trigger.hasAttribute('tabindex')) {
 trigger.setAttribute('tabindex', '0');
 }

 // Add keyboard event
 trigger.addEventListener('keydown', event => {
 if (event.key === 'Enter' || event.key === ' ') {
 event.preventDefault();
 toggleDropdown(trigger, target);
 } else if (event.key === 'Escape' && trigger.getAttribute('aria-expanded') ===
 event.preventDefault();
 hideDropdown(trigger, target);
 }
 });
});

// Close dropdowns when clicking outside
document.addEventListener('click', event => {
 const openTriggers = document.querySelectorAll('[data-dropdown][aria-expanded="true"]

 openTriggers.forEach(trigger => {
 const targetId = trigger.getAttribute('data-dropdown');
 const target = document.getElementById(targetId);

 if (target && !target.contains(event.target)) {
 hideDropdown(trigger, target);
 }
 });
});

} catch (error) {
 console.error('Error initializing dropdowns:', error);
}

}

/***
 * Toggle dropdown
 * @param {Element} trigger - Trigger element
 * @param {Element} target - Target element
 */
function toggleDropdown(trigger, target) {
 try {

```

```

// Check current state
const isExpanded = trigger.getAttribute('aria-expanded') === 'true';

if (isExpanded) {
 hideDropdown(trigger, target);
} else {
 showDropdown(trigger, target);
}
} catch (error) {
 console.error('Error toggling dropdown:', error);
}

/**
 * Show dropdown
 * @param {Element} trigger - Trigger element
 * @param {Element} target - Target element
 */
function showDropdown(trigger, target) {
 try {
 // Hide all other dropdowns
 const openTriggers = document.querySelectorAll('[data-dropdown][aria-expanded="true"]')

 openTriggers.forEach(openTrigger => {
 if (openTrigger !== trigger) {
 const openTargetId = openTrigger.getAttribute('data-dropdown');
 const openTarget = document.getElementById(openTargetId);

 if (openTarget) {
 hideDropdown(openTrigger, openTarget);
 }
 }
 });
 }

 // Position dropdown
 const rect = trigger.getBoundingClientRect();

 target.style.top = `${rect.bottom + window.scrollY}px`;
 target.style.left = `${rect.left + window.scrollX}px`;
 target.style.minWidth = `${rect.width}px`;

 // Show dropdown
 target.style.display = 'block';
 trigger.setAttribute('aria-expanded', 'true');
}

```

```

// Focus first item
const firstItem = target.querySelector('a, button, [tabindex]');

if (firstItem) {
 firstItem.focus();
}
} catch (error) {
 console.error('Error showing dropdown:', error);
}

/**
 * Hide dropdown
 * @param {Element} trigger - Trigger element
 * @param {Element} target - Target element
 */
function hideDropdown(trigger, target) {
 try {
 // Hide dropdown
 target.style.display = 'none';
 trigger.setAttribute('aria-expanded', 'false');

 // Focus trigger
 trigger.focus();
 } catch (error) {
 console.error('Error hiding dropdown:', error);
 }
}

/**
 * Initialize form UI enhancements
 */
function initializeFormUI() {
 try {
 // Initialize form validation
 initializeFormValidation();

 // Initialize form range inputs
 initializeRangeInputs();

 // Add image to PDF
 doc.addImage(imgData, 'PNG', 15, y + 5, imgWidth, imgHeight);

 y += imgHeight + 15;
 }
}

```

```

 }

 // Add chart description if available
 if (chart.description) {
 // Check if need to add new page
 if (y > 240) {
 doc.addPage();
 y = 20;
 }

 doc.setFont('helvetica', 'normal');
 doc.setFontSize(10);

 const lines = doc.splitTextToSize(chart.description, 180);
 doc.text(lines, 15, y);

 y += lines.length * 5 + 10;
 }
}

// Add disclaimer to last page
await this.#addDisclaimer(doc);
} catch (error) {
 console.error('Error adding charts to PDF:', error);
}
}

export default PDFGenerator;

```

## 5. Core Application Files

### 5.1 main.js

javascript

```
/**
 * Main Application for CVD Risk Toolkit
 *
 * Initializes all components and modules, sets up event listeners,
 * and manages the application lifecycle.
 */

// Import core utilities
import ErrorDetectionSystem from './utils/error-detection-system.js';
import RuntimeProtection from './utils/runtime-protection.js';
import EventBus from './utils/event-bus.js';
import DeviceCapabilityDetector from './utils/device-capability-detector.js';
import ModuleLoader from './utils/module-loader.js';
import SecureStorage from './utils/secure-storage.js';

// Import UI modules
import TabManager from './utils/tab-manager.js';
import LoadingManager from './utils/loading-manager.js';
import XSSProtection from './utils/xss-protection.js';

// Initialize application
document.addEventListener('DOMContentLoaded', initializeApp);

/**
 * Initialize the application
 */
async function initializeApp() {
 try {
 console.log('Initializing CVD Risk Toolkit...');

 // Show Loading indicator
 LoadingManager.showLoading('Initializing application...');

 // Set up error handling
 ErrorDetectionSystem.initErrorTracking();

 // Detect device capabilities
 const capabilities = DeviceCapabilityDetector.detect();
 DeviceCapabilityDetector.applyOptimizations();

 // Initialize secure storage
 await SecureStorage.initialize();
 } catch (error) {
 console.error(`An error occurred during initialization: ${error.message}`);
 }
}
})();
```

```
// Set up tab navigation
initializeTabNavigation();

// Initialize event listening
initializeEventListeners();

// Initialize content security policy
initializeCSP();

// PreLoad critical modules
await ModuleLoader.preloadCriticalModules();

// Initialize calculator forms
await initializeForms();

// Set up offline detection
initializeOfflineDetection();

// Make app visible once initialized
document.body.classList.add('initialized');

// Hide Loading indicator
LoadingManager.hideLoading();

 console.log('CVD Risk Toolkit initialized successfully');
} catch (error) {
 console.error('Error initializing application:', error);

// Show error notification
ErrorDetectionSystem.showCriticalErrorPage(
 new Error('Failed to initialize application. Please refresh the page or try again !'));
}

// Hide Loading indicator
LoadingManager.hideLoading();
}

}

/**
 * Initialize tab navigation
*/
function initializeTabNavigation() {
 try {
 // Create tab manager instance
```

```

window.tabManager = new TabManager({
 tabSelector: '.tab-button',
 contentSelector: '.tab-content',
 activeClass: 'active',
 onChange: handleTabChange,
 persistState: true
});

// Enable keyboard navigation
document.addEventListener('keydown', (event) => {
 if (event.ctrlKey || event.metaKey) {
 // Ctrl/Cmd + number keys for tab navigation
 if (event.key >= '1' && event.key <= '5') {
 event.preventDefault();
 const tabIndex = parseInt(event.key) - 1;
 const tabs = document.querySelectorAll('.tab-button');
 if (tabs[tabIndex]) {
 tabs[tabIndex].click();
 }
 }
 }
});
} catch (error) {
 console.error('Error initializing tab navigation:', error);
}
}

/**
 * Handle tab change event
 * @param {number} index - Tab index
 * @param {Element} tab - Tab element
 * @param {Element} content - Content element
 */
async function handleTabChange(index, tab, content) {
 try {
 // Get tab ID
 const tabId = tab.getAttribute('data-tab');

 // Load tab-specific modules
 const viewModules = await ModuleLoader.loadViewModules(tabId);

 // Initialize tab-specific content
 if (tabId === 'frs' && viewModules.framinghamAlgorithm) {
 // Initialize FRS calculator
 }
 }
}

```

```

 initializeFRSCalculator(viewModules.framinghamAlgorithm);
 } else if (tabId === 'qrisk' && viewModules.qrisk3Algorithm) {
 // Initialize QRISK3 calculator
 initializeQRISK3Calculator(viewModules.qrisk3Algorithm);
 } else if (tabId === 'combined' && viewModules.combinedViewManager) {
 // Initialize combined view
 viewModules.combinedViewManager.initialize();
 } else if (tabId === 'medication' && viewModules.treatmentRecommendations) {
 // Initialize medication recommendations
 initializeMedicationRecommendations(viewModules.treatmentRecommendations);
 } else if (tabId === 'visualization' && viewModules.chartRenderer) {
 // Initialize visualizations
 initializeVisualizations(viewModules.chartRenderer);
 }
} catch (error) {
 console.error('Error handling tab change:', error);

 // Show error notification
 ErrorDetectionSystem.showErrorNotification(
 new Error(`Failed to initialize ${tab.textContent.trim()} tab. Some features may be
 'Tab Initialization Error'
);
}

}

/***
 * Initialize event listeners
 */
function initializeEventListeners() {
 try {
 // Set up global event listeners
 document.addEventListener('click', (event) => {
 // Handle help button clicks
 if (event.target.closest('.help-button')) {
 const helpButton = event.target.closest('.help-button');
 const helpId = helpButton.getAttribute('data-help-id');

 if (helpId) {
 showHelp(helpId);
 }
 }

 // Handle export button clicks
 if (event.target.closest('.export-button')) {

```

```

 const exportButton = event.target.closest('.export-button');
 const exportFormat = exportButton.getAttribute('data-format') || 'pdf';
 const exportId = exportButton.getAttribute('data-export-id');

 if (exportId) {
 exportData(exportId, exportFormat);
 }
 }
});

// Set up form submission prevention
document.addEventListener('submit', (event) => {
 // Prevent actual form submissions
 event.preventDefault();

 // Process form data
 const form = event.target;
 processForm(form);
});

// Subscribe to EventBus events
EventBus.subscribe('calculator-updated', (data) => {
 updateCalculatorUI(data);
});

EventBus.subscribe('recommendation-updated', (data) => {
 updateRecommendationsUI(data);
});

} catch (error) {
 console.error('Error initializing event listeners:', error);
}

}

/***
 * Initialize Content Security Policy
 */
function initializeCSP() {
 try {
 // Set up CSP via meta tag if not already set by server
 if (!document.querySelector('meta[http-equiv="Content-Security-Policy"]')) {
 const cspMeta = document.createElement('meta');
 cspMeta.httpEquiv = 'Content-Security-Policy';
 cspMeta.content =
 "default-src 'self'; " +

```

```
 "script-src 'self' 'unsafe-inline' https://cdnjs.cloudflare.com; " +
 "style-src 'self' 'unsafe-inline' https://cdnjs.cloudflare.com; " +
 "img-src 'self' data: blob:; " +
 "connect-src 'self'; " +
 "font-src 'self' https://cdnjs.cloudflare.com; " +
 "object-src 'none'; " +
 "frame-src 'none'; " +
 "base-uri 'self'; " +
 "form-action 'none';";

 document.head.appendChild(cspMeta);
 }
} catch (error) {
 console.error('Error initializing CSP:', error);
}
}

/***
 * Initialize forms
*/
async function initializeForms() {
 try {
 // Get all calculator forms
 const forms = document.querySelectorAll('form[data-calculator]');

 // Initialize each form
 for (const form of forms) {
 // Get calculator type
 const calculatorType = form.getAttribute('data-calculator');

 // Set up form validation
 XSSProtection.protectForm(form);

 // Initialize form elements
 initializeFormElements(form);

 // Set up auto-calculation
 setupAutoCalculation(form, calculatorType);
 }
 } catch (error) {
 console.error('Error initializing forms:', error);
 }
}
```

```
/**
 * Initialize form elements
 * @param {HTMLFormElement} form - Form element
 */

function initializeFormElements(form) {
 try {
 // Initialize select elements
 const selects = form.querySelectorAll('select');
 selects.forEach(select => {
 // Try to restore saved value from LocalStorage
 const savedValue = localStorage.getItem(`#${form.id}-${select.id}`);
 if (savedValue) {
 select.value = savedValue;
 }

 // Save value to LocalStorage on change
 select.addEventListener('change', () => {
 localStorage.setItem(`#${form.id}-${select.id}`, select.value);
 });
 });

 // Initialize number inputs
 const numberInputs = form.querySelectorAll('input[type="number"]');
 numberInputs.forEach(input => {
 // Try to restore saved value from LocalStorage
 const savedValue = localStorage.getItem(`#${form.id}-${input.id}`);
 if (savedValue) {
 input.value = savedValue;
 }

 // Save value to LocalStorage on change
 input.addEventListener('change', () => {
 localStorage.setItem(`#${form.id}-${input.id}`, input.value);
 });

 // Add validation
 input.addEventListener('input', validateNumberInput);
 });

 // Initialize checkboxes
 const checkboxes = form.querySelectorAll('input[type="checkbox"]');
 checkboxes.forEach(checkbox => {
 // Try to restore saved value from LocalStorage
 const savedValue = localStorage.getItem(`#${form.id}-${checkbox.id}`);
```

```

 if (savedValue !== null) {
 checkbox.checked = savedValue === 'true';
 }

 // Save value to LocalStorage on change
 checkbox.addEventListener('change', () => {
 localStorage.setItem(`#${form.id}-${checkbox.id}`, checkbox.checked);
 });
 });

 // Initialize radio buttons
 const radioGroups = {};
 const radios = form.querySelectorAll('input[type="radio"]');
 radios.forEach(radio => {
 // Group radios by name
 if (!radioGroups[radio.name]) {
 radioGroups[radio.name] = [];
 }
 radioGroups[radio.name].push(radio);

 // Save value to LocalStorage on change
 radio.addEventListener('change', () => {
 if (radio.checked) {
 localStorage.setItem(`#${form.id}-${radio.name}`, radio.value);
 }
 });
 });

 // Try to restore saved radio values
 for (const [name, group] of Object.entries(radioGroups)) {
 const savedValue = localStorage.getItem(`#${form.id}-${name}`);
 if (savedValue) {
 group.forEach(radio => {
 radio.checked = radio.value === savedValue;
 });
 }
 }
} catch (error) {
 console.error('Error initializing form elements:', error);
}

/**
 * Validate number input

```

```

* @param {Event} event - Input event
*/
function validateNumberInput(event) {
 const input = event.target;
 const value = input.value;

 // Clear validation error
 input.setCustomValidity('');
 input.classList.remove('invalid');

 // Skip empty values
 if (!value) return;

 // Get min/max values
 const min = parseFloat(input.getAttribute('min') || '-Infinity');
 const max = parseFloat(input.getAttribute('max') || 'Infinity');

 // Check if value is a number
 if (isNaN(value)) {
 input.setCustomValidity('Please enter a valid number');
 input.classList.add('invalid');
 return;
 }

 // Check if value is within range
 if (value < min) {
 input.setCustomValidity(`Value must be at least ${min}`);
 input.classList.add('invalid');
 } else if (value > max) {
 input.setCustomValidity(`Value must be at most ${max}`);
 input.classList.add('invalid');
 }

 // Check if field has custom validation
 const validationType = input.getAttribute('data-validation');
 if (validationType) {
 performCustomValidation(input, validationType, value);
 }
}

/**
 * Perform custom validation
 * @param {HTMLInputElement} input - Input element
 * @param {string} validationType - Validation type

```

```
* @param {string} value - Input value
*/
function performCustomValidation(input, validationType, value) {
 try {
 const numValue = parseFloat(value);

 // Skip if not a number
 if (isNaN(numValue)) return;

 switch (validationType) {
 case 'systolicBP':
 // Check for physiologically plausible BP range
 if (numValue < 60) {
 input.setCustomValidity('Systolic BP is unusually low. Please check the val
 input.classList.add('warning');
 } else if (numValue > 250) {
 input.setCustomValidity('Systolic BP is unusually high. Please check the va
 input.classList.add('warning');
 }
 break;

 case 'diastolicBP':
 // Check for physiologically plausible BP range
 if (numValue < 40) {
 input.setCustomValidity('Diastolic BP is unusually low. Please check the va
 input.classList.add('warning');
 } else if (numValue > 150) {
 input.setCustomValidity('Diastolic BP is unusually high. Please check the v
 input.classList.add('warning');
 }
 break;

 case 'cholesterol':
 // Check for unit and physiologically plausible range
 const unit = document.querySelector('input[name="cholesterol-unit"]:checked')?

 if (unit === 'mmol/L') {
 if (numValue < 1.0) {
 input.setCustomValidity('Cholesterol value is unusually low. Please che
 input.classList.add('warning');
 } else if (numValue > 15.0) {
 input.setCustomValidity('Cholesterol value is unusually high. Please ch
 input.classList.add('warning');
 }
 }
 }
 }
}
```

```

 } else if (unit === 'mg/dL') {
 if (numValue < 40) {
 input.setCustomValidity('Cholesterol value is unusually low. Please check');
 input.classList.add('warning');
 } else if (numValue > 600) {
 input.setCustomValidity('Cholesterol value is unusually high. Please check');
 input.classList.add('warning');
 }
 }
 break;

 case 'hdl':
 // Check for unit and physiologically plausible range
 const hdlUnit = document.querySelector('input[name="cholesterol-unit"]:checked')

 if (hdlUnit === 'mmol/L') {
 if (numValue < 0.5) {
 input.setCustomValidity('HDL value is unusually low. Please check the value');
 input.classList.add('warning');
 } else if (numValue > 3.0) {
 input.setCustomValidity('HDL value is unusually high. Please check the value');
 input.classList.add('warning');
 }
 } else if (hdlUnit === 'mg/dL') {
 if (numValue < 20) {
 input.setCustomValidity('HDL value is unusually low. Please check the value');
 input.classList.add('warning');
 } else if (numValue > 120) {
 input.setCustomValidity('HDL value is unusually high. Please check the value');
 input.classList.add('warning');
 }
 }
 break;
 }

} catch (error) {
 console.error('Error performing custom validation:', error);
}

}

/**
 * Set up auto calculation for form
 * @param {HTMLFormElement} form - Form element
 * @param {string} calculatorType - Calculator type
 */

```

```
function setupAutoCalculation(form, calculatorType) {
 try {
 // Add event listeners to all form elements
 const formElements = form.elements;

 for (let i = 0; i < formElements.length; i++) {
 const element = formElements[i];

 // Skip buttons
 if (element.type === 'button' || element.type === 'submit') {
 continue;
 }

 // Create debounced event handler
 const debouncedHandler = RuntimeProtection.debounce(() => {
 // Validate form before calculation
 if (form.checkValidity()) {
 calculateRisk(form, calculatorType);
 }
 }, 500);

 // Add event listener
 element.addEventListener('change', () => {
 // Publish event for cross-tab sync
 EventBus.publish('form-field-changed', {
 formId: form.id,
 fieldId: element.id,
 value: element.type === 'checkbox' ? element.checked : element.value
 });
 });

 // Trigger calculation if form is valid
 debouncedHandler();
 });
 }

 // Setup calculation button
 const calculateButton = form.querySelector('.calculate-button');
 if (calculateButton) {
 calculateButton.addEventListener('click', () => {
 // Check form validity
 if (form.reportValidity()) {
 calculateRisk(form, calculatorType);
 }
 });
 }
}
```

```

 }

 } catch (error) {
 console.error('Error setting up auto calculation:', error);
 }
}

/***
 * Calculate cardiovascular risk
 * @param {HTMLFormElement} form - Form element
 * @param {string} calculatorType - Calculator type
 */
async function calculateRisk(form, calculatorType) {
 try {
 // Show Loading indicator
 LoadingManager.showLoading('Calculating risk...');

 // Get form data
 const formData = new FormData(form);
 const data = {};

 // Convert form data to object
 for (const [key, value] of formData.entries()) {
 data[key] = value;
 }

 // Add checkbox values (unchecked checkboxes are not included in FormData)
 const checkboxes = form.querySelectorAll('input[type="checkbox"]');
 checkboxes.forEach(checkbox => {
 data[checkbox.name] = checkbox.checked;
 });

 // Load calculator module
 let Calculator;

 if (calculatorType === 'frs') {
 const module = await ModuleLoader.loadModule('./calculations/framingham-algorithm.js');
 Calculator = module;
 } else if (calculatorType === 'qrisk') {
 const module = await ModuleLoader.loadModule('./calculations/qrisk3-algorithm.js');
 Calculator = module;
 } else {
 throw new Error(`Unknown calculator type: ${calculatorType}`);
 }
 }
}

```

```

 // Calculate risk
 const result = Calculator.calculateRisk(data);

 // Store result in session storage
 sessionStorage.setItem(`#${calculatorType}_result`, JSON.stringify(result));

 // Update UI
 updateResultsUI(result, calculatorType);

 // Generate recommendations
 generateRecommendations(data, result, calculatorType);

 // Publish calculation event
 EventBus.publish('risk-calculated', {
 calculator: calculatorType,
 result: result
 });

 // Hide Loading indicator
 LoadingManager.hideLoading();
} catch (error) {
 console.error('Error calculating risk:', error);

 // Show error notification
 ErrorDetectionSystem.showErrorNotification(
 new Error(`Error calculating risk: ${error.message}`),
 'Calculation Error'
);
}

// Hide Loading indicator
LoadingManager.hideLoading();
}

}

/**
 * Update results UI
 * @param {Object} result - Calculation result
 * @param {string} calculatorType - Calculator type
 */
function updateResultsUI(result, calculatorType) {
 try {
 // Get results container
 const resultsContainer = document.getElementById(`#${calculatorType}-results`);

```

```
if (!resultsContainer) return;

// Clear previous results
resultsContainer.innerHTML = '';

// Check if calculation was successful
if (!result.success) {
 // Show error message
 const errorMessage = document.createElement('div');
 errorMessage.className = 'error-message';
 errorMessage.textContent = result.error || 'An error occurred during calculation';
 resultsContainer.appendChild(errorMessage);
 return;
}

// Create results HTML
let resultsHTML = '';

// Add risk percentage
const riskPercent = result.modifiedRiskPercent || result.tenYearRiskPercent;
resultsHTML += `
 <div class="risk-percentage-container">
 <div class="risk-percentage ${result.riskCategory}">${riskPercent}%</div>
 <div class="risk-label">10-Year CVD Risk</div>
 </div>
 <div class="risk-category-container">
 <div class="risk-category-label">Risk Category:</div>
 <div class="risk-category-value ${result.riskCategory}">
 ${result.riskCategory.charAt(0).toUpperCase() + result.riskCategory.slice(1)}
 </div>
 </div>
`;

// Add additional information based on calculator type
if (calculatorType === 'qrisk' && result.heartAge) {
 resultsHTML += `
 <div class="additional-results">
 <div class="result-row">
 <div class="result-label">Heart Age:</div>
 <div class="result-value">${result.heartAge} years</div>
 </div>
 <div class="result-row">
 <div class="result-label">Relative Risk:</div>
 <div class="result-value">${result.relativeRisk}x</div>
 </div>
 </div>
 `;
}
```

```

 </div>
 </div>
 `;
} else if (calculatorType === 'frs' && result.riskModifiers?.length > 0) {
 resultsHTML += `
 <div class="additional-results">
 <div class="result-row">
 <div class="result-label">Risk Modifiers:</div>
 <div class="result-value">
 <ul class="risk-modifiers">
 ${result.riskModifiers.map(modifier =>
 ${modifier.factor} (${modifier.multiplier}x)
).join('')}

 </div>
 </div>
 </div>
 `;
}

// Add calculation date
const calculationDate = new Date(result.calculationDate).toLocaleString();
resultsHTML += `
 <div class="calculation-date">
 Calculated on: ${calculationDate}
 </div>
 <div class="action-buttons">
 <button type="button" class="btn btn-secondary export-button" data-format="pdf">
 <i class="icon-download"></i> Export PDF
 </button>
 </div>
`;
}

// Update results container
resultsContainer.innerHTML = resultsHTML;

// Show results container
resultsContainer.style.display = 'block';
} catch (error) {
 console.error('Error updating results UI:', error);
}
}

/**

```

```
* Generate treatment recommendations
* @param {Object} data - Patient data
* @param {Object} result - Calculation result
* @param {string} calculatorType - Calculator type
*/
async function generateRecommendations(data, result, calculatorType) {
 try {
 // Skip if calculation failed
 if (!result.success) return;

 // Load recommendations module
 const TreatmentRecommendations = (await ModuleLoader.loadModule('./calculations/treatme

 // Get recommendation method based on calculator type
 let recommendationMethod;

 if (calculatorType === 'frs') {
 recommendationMethod = TreatmentRecommendations.generateFRSRecommendations;
 } else if (calculatorType === 'qrisk') {
 recommendationMethod = TreatmentRecommendations.generateQRISKRecommendations;
 } else {
 return;
 }

 // Generate recommendations
 const recommendations = await recommendationMethod(data, result);

 // Skip if recommendations generation failed
 if (!recommendations || !recommendations.success) return;

 // Store recommendations in session storage
 sessionStorage.setItem(`"${calculatorType}"_recommendations`, JSON.stringify(recommendati

 // Update recommendations UI
 updateRecommendationsUI(recommendations, calculatorType);

 // Publish recommendations event
 EventBus.publish('recommendation-generated', {
 calculator: calculatorType,
 recommendations: recommendations
 });
 } catch (error) {
 console.error('Error generating recommendations:', error);
 }
}
```

```

}

/**
 * Update recommendations UI
 * @param {Object} recommendations - Recommendations
 * @param {string} calculatorType - Calculator type
 */
function updateRecommendationsUI(recommendations, calculatorType) {
 try {
 // Get recommendations container
 const recommendationsContainer = document.getElementById(`#${calculatorType}-recommendat
 if (!recommendationsContainer) return;

 // Update recommendations container
 recommendationsContainer.innerHTML = recommendations.summary || '';
 // Show recommendations container
 recommendationsContainer.style.display = 'block';
 } catch (error) {
 console.error('Error updating recommendations UI:', error);
 }
 }

 /**
 * Initialize Framingham Risk Score calculator
 * @param {Object} Calculator - Framingham calculator module
 */
 function initializeFRSCalculator(Calculator) {
 // Implemented in the form initialization section
 }

 /**
 * Initialize QRISK3 calculator
 * @param {Object} Calculator - QRISK3 calculator module
 */
 function initializeQRISK3Calculator(Calculator) {
 // Implemented in the form initialization section
 }

 /**
 * Initialize medication recommendations
 * @param {Object} TreatmentRecommendations - Treatment recommendations module
 */

```

```
function initializeMedicationRecommendations(TreatmentRecommendations) {
 try {
 // Get medication form
 const medicationForm = document.getElementById('medication-form');

 if (!medicationForm) return;

 // Set up form submission
 medicationForm.addEventListener('submit', async (event) => {
 event.preventDefault();

 // Show Loading indicator
 LoadingManager.showLoading('Generating recommendations...');

 // Get form data
 const formData = new FormData(medicationForm);
 const data = {};

 // Convert form data to object
 for (const [key, value] of formData.entries()) {
 data[key] = value;
 }

 // Add checkbox values
 const checkboxes = medicationForm.querySelectorAll('input[type="checkbox"]');
 checkboxes.forEach(checkbox => {
 data[checkbox.name] = checkbox.checked;
 });

 try {
 // Get saved results if available
 let frsResult = null;
 let qriskResult = null;

 try {
 const frsResultJson = sessionStorage.getItem('frs_result');
 if (frsResultJson) {
 frsResult = JSON.parse(frsResultJson);
 }
 }

 const qriskResultJson = sessionStorage.getItem('qrisk_result');
 if (qriskResultJson) {
 qriskResult = JSON.parse(qriskResultJson);
 }
 }
 });
 }
}
```

```
 } catch (e) {
 console.error('Error retrieving saved results:', e);
 }

 // Generate recommendations
 const recommendations = await TreatmentRecommendations.generateRecommendations(
 data,
 frsResult,
 qriskResult
);

 // Update recommendations UI
 const recommendationsContainer = document.getElementById('medication-recommendations');

 if (recommendationsContainer) {
 recommendationsContainer.innerHTML = recommendations.summary || '';
 recommendationsContainer.style.display = 'block';
 }

 // Store recommendations in session storage
 sessionStorage.setItem('recommendations', JSON.stringify(recommendations));

 // Publish recommendations event
 EventBus.publish('recommendation-generated', {
 calculator: 'medication',
 recommendations: recommendations
 });
 } catch (error) {
 console.error('Error generating medication recommendations:', error);

 // Show error notification
 ErrorDetectionSystem.showErrorNotification(
 new Error(`Error generating recommendations: ${error.message}`),
 'Recommendations Error'
);
 }

 // Hide Loading indicator
 LoadingManager.hideLoading();
});

} catch (error) {
 console.error('Error initializing medication recommendations:', error);
}
}
```

```

/**
 * Initialize visualizations
 * @param {Object} ChartRenderer - Chart renderer module
 */
function initializeVisualizations(ChartRenderer) {
 try {
 // Get visualization form
 const visualizationForm = document.getElementById('visualization-form');

 if (!visualizationForm) return; // If value is exactly one of the status codes, return
 if (['non', 'ex', 'light', 'moderate', 'heavy'].includes(normalizedValue)) {
 return normalizedValue;
 }

 // Default to moderate if it's a confirmed smoker but level unknown
 if (this.#mapBooleanValue(value) === true) {
 return 'moderate';
 }

 // Return original value if mapping not found
 return value;
 }

 /**
 * Map diabetes status value
 * @param {*} value - Value to map
 * @param {string} fromSystem - Source system
 * @param {string} toSystem - Target system
 * @returns {string} - Mapped diabetes status value
 * @private
 */
 static #mapDiabetesStatusValue(value, fromSystem, toSystem) {
 if (!value) return null;

 // Define mapping tables
 const commonMapping = {
 // Common diabetes status values
 'none': ['none', 'no', 'no diabetes', 'non-diabetic', '0', 'n', 'false'],
 'type1': ['type1', 'type 1', 't1', 't1dm', 'type1 diabetes', 'type 1 diabetes', '1'],
 'type2': ['type2', 'type 2', 't2', 't2dm', 'type2 diabetes', 'type 2 diabetes', '2']
 };

 // System-specific mappings
 }
}

```

```

const systemMappings = {
 epic: {
 'none': ['0', 'N', 'NO', 'NON-DIABETIC'],
 'type1': ['T1DM', 'TYPE 1', 'IDDM'],
 'type2': ['T2DM', 'TYPE 2', 'NIDDM']
 },
 cerner: {
 'none': ['NON_DIABETIC', '0'],
 'type1': ['DM_TYPE_1', 'IDDM', '1'],
 'type2': ['DM_TYPE_2', 'NIDDM', '2']
 }
};

// Convert to lowercase if string
const normalizedValue = typeof value === 'string' ? value.toLowerCase() : value;

// Try to find match in system-specific mapping
if (fromSystem in systemMappings) {
 for (const [status, values] of Object.entries(systemMappings[fromSystem])) {
 if (values.map(v => v.toLowerCase()).includes(normalizedValue)) {
 return status;
 }
 }
}

// Try to find match in common mapping
for (const [status, values] of Object.entries(commonMapping)) {
 if (values.includes(normalizedValue)) {
 return status;
 }
}

// If value is exactly one of the status codes, return it
if (['none', 'type1', 'type2'].includes(normalizedValue)) {
 return normalizedValue;
}

// Default to type2 if it's confirmed diabetes but type unknown
if (this.#mapBooleanValue(value) === true) {
 return 'type2';
}

// Return original value if mapping not found
return value;

```

```

}

/**
 * Map ethnicity value
 * @param {*} value - Value to map
 * @param {string} fromSystem - Source system
 * @param {string} toSystem - Target system
 * @returns {string} - Mapped ethnicity value
 * @private
 */
static #mapEthnicityValue(value, fromSystem, toSystem) {
 if (!value) return null;

 // QRISK3 ethnicity values
 const qrisk3Ethnicities = {
 'white': ['white', 'caucasian', 'european', '1'],
 'indian': ['indian', 'south asian indian', '2'],
 'pakistani': ['pakistani', 'south asian pakistani', '3'],
 'bangladeshi': ['bangladeshi', 'south asian bangladeshi', '4'],
 'other_asian': ['other_asian', 'other asian', 'asian other', 'east asian', '5'],
 'black_caribbean': ['black_caribbean', 'black caribbean', 'afro-caribbean', '6'],
 'black_african': ['black_african', 'black african', 'african', '7'],
 'chinese': ['chinese', '8'],
 'other': ['other', 'mixed', 'other ethnic group', '9']
 };

 // Convert to lowercase if string
 const normalizedValue = typeof value === 'string' ? value.toLowerCase() : value;

 // If target system is QRISK3, map to QRISK3 ethnicity values
 if (toSystem === 'qrisk3') {
 for (const [ethnicity, values] of Object.entries(qrisk3Ethnicities)) {
 if (values.includes(normalizedValue)) {
 return ethnicity;
 }
 }
 }

 // Default to white if mapping not found
 return 'white';
}

// If source system is QRISK3, map from QRISK3 ethnicity values
if (fromSystem === 'qrisk3') {
 for (const [ethnicity, values] of Object.entries(qrisk3Ethnicities)) {

```

```

 if (ethnicity === normalizedValue || values.includes(normalizedValue)) {
 return ethnicity.replace('_', ' ');
 }
 }

 // Return original value if mapping not found
 return value;
}

/**
 * Map unit conversion
 * @param {string} fieldName - Field name
 * @param {number} value - Value to convert
 * @param {string} fromSystem - Source system
 * @param {string} toSystem - Target system
 * @returns {number} - Converted value
 * @private
 */
static #mapUnitConversion(fieldName, value, fromSystem, toSystem) {
 if (value === null || value === undefined || isNaN(value)) {
 return value;
 }

 // Convert to number if string
 const numValue = typeof value === 'string' ? parseFloat(value) : value;

 // Define unit systems
 const unitSystems = {
 // SI units
 metric: {
 height: 'cm',
 weight: 'kg',
 totalCholesterol: 'mmol/L',
 hdl: 'mmol/L',
 ldl: 'mmol/L',
 triglycerides: 'mmol/L'
 },
 // US units
 us: {
 height: 'in',
 weight: 'lb',
 totalCholesterol: 'mg/dL',
 hdl: 'mg/dL',
 ldl: 'mg/dL',
 triglycerides: 'mg/dL'
 }
 };
}

```

```

 ldl: 'mg/dL',
 triglycerides: 'mg/dL'
 }
};

// Define system unit maps
const systemUnitMap = {
 'epic': 'us',
 'cerner': 'us',
 'framingham': 'metric',
 'qrisk3': 'metric'
};

// Get unit systems for source and target
const fromUnitSystem = systemUnitMap[fromSystem] || 'metric';
const toUnitSystem = systemUnitMap[toSystem] || 'metric';

// Skip if unit systems are the same
if (fromUnitSystem === toUnitSystem) {
 return numValue;
}

// Conversion factors
const conversionFactors = {
 // Height: cm to in
 heightToUS: 0.393701,
 // Height: in to cm
 heightToMetric: 2.54,
 // Weight: kg to lb
 weightToUS: 2.20462,
 // Weight: lb to kg
 weightToMetric: 0.453592,
 // Cholesterol: mmol/L to mg/dL
 cholesterolToUS: 38.67,
 // Cholesterol: mg/dL to mmol/L
 cholesterolToMetric: 0.02586,
 // Triglycerides: mmol/L to mg/dL
 triglyceridesToUS: 88.5,
 // Triglycerides: mg/dL to mmol/L
 triglyceridesToMetric: 0.01129
};

// Convert based on field name and unit systems
if (fromUnitSystem === 'metric' && toUnitSystem === 'us') {

```

```

 // Convert from metric to US
 switch (fieldName) {
 case 'height':
 return numValue * conversionFactors.heightToUS;
 case 'weight':
 return numValue * conversionFactors.weightToUS;
 case 'totalCholesterol':
 case 'hdl':
 case 'ldl':
 return numValue * conversionFactors.cholesterolToUS;
 case 'triglycerides':
 return numValue * conversionFactors.triglyceridesToUS;
 }
} else if (fromUnitSystem === 'us' && toUnitSystem === 'metric') {
 // Convert from US to metric
 switch (fieldName) {
 case 'height':
 return numValue * conversionFactors.heightToMetric;
 case 'weight':
 return numValue * conversionFactors.weightToMetric;
 case 'totalCholesterol':
 case 'hdl':
 case 'ldl':
 return numValue * conversionFactors.cholesterolToMetric;
 case 'triglycerides':
 return numValue * conversionFactors.triglyceridesToMetric;
 }
}

// Return original value if no conversion defined
return numValue;
}

export default FieldMapper;

```

## 4.4 pdf-generator.js

javascript

```
/**
 * PDF Generator for CVD Risk Toolkit
 *
 * Generates PDF reports of risk assessment results, charts,
 * and treatment recommendations.
 */

class PDFGenerator {
 /**
 * Generate report as PDF
 * @param {Object} data - Report data
 * @returns {Promise<Object>} - Generation result
 */
 static async generateReport(data) {
 try {
 console.log('Generating PDF report...');

 // Try to import jsPDF library
 const jsPDF = (await import('jspdf')).default;
 const autoTable = (await import('jspdf-autotable')).default;

 // Create new PDF document
 const doc = new jsPDF({
 orientation: 'portrait',
 unit: 'mm',
 format: 'a4'
 });

 // Add report title
 const title = 'Cardiovascular Risk Assessment Report';
 doc.setFont('helvetica', 'bold');
 doc.setFontSize(18);
 doc.text(title, 105, 20, { align: 'center' });

 // Add generation date
 const dateStr = new Date().toLocaleDateString();
 doc.setFont('helvetica', 'normal');
 doc.setFontSize(10);
 doc.text(`Generated on: ${dateStr}`, 105, 27, { align: 'center' });

 // Add patient information section
 await this.#addPatientInformation(doc, data, 35);
 } catch (error) {
 console.error('Error generating PDF report:', error);
 }
 }
}
// Example usage:
const generator = new PDFGenerator();
generator.generateReport(
 {
 patientName: 'John Doe',
 age: 45,
 gender: 'Male',
 riskScore: 120,
 cholesterol: 200,
 bloodPressure: 140/90,
 smokingStatus: 'Non-smoker',
 exerciseLevel: 'Low',
 dietType: 'Standard',
 familyHistory: false,
 treatmentPlan: 'Lifestyle changes'
 }
)
```

```

 // Add risk assessment section
 await this.#addRiskAssessment(doc, data, 80);

 // Add recommendations section
 await this.#addRecommendations(doc, data, 150);

 // Add disclaimer
 await this.#addDisclaimer(doc);

 // Add charts if available
 if (data.charts) {
 await this.#addCharts(doc, data.charts);
 }

 // Create filename
 const timestamp = new Date().toISOString().replace(/[-.:]/g, '').substring(0, 15);
 const filename = `cvd-risk-report-${timestamp}.pdf`;

 // Save PDF
 doc.save(filename);

 return {
 success: true,
 filename,
 format: 'pdf'
 };
} catch (error) {
 console.error('Error generating PDF report:', error);

 return {
 success: false,
 error: error.message || 'Unknown error generating PDF',
 format: 'pdf'
 };
}

/**
 * Generate chart PDF
 * @param {HTMLCanvasElement} canvas - Chart canvas
 * @param {string} details - Chart details HTML
 * @returns {Promise<Object>} - Generation result
 */
static async generateChartPDF(canvas, details) {

```

```
try {
 console.log('Generating chart PDF...');

 // Try to import jsPDF library
 const jsPDF = (await import('jspdf')).default;

 // Create new PDF document
 const doc = new jsPDF({
 orientation: 'landscape',
 unit: 'mm',
 format: 'a4'
 });

 // Add title
 const title = 'Cardiovascular Risk Chart';
 doc.setFont('helvetica', 'bold');
 doc.setFontSize(18);
 doc.text(title, 149, 20, { align: 'center' });

 // Add generation date
 const dateStr = new Date().toLocaleDateString();
 doc.setFont('helvetica', 'normal');
 doc.setFontSize(10);
 doc.text(`Generated on: ${dateStr}`, 149, 27, { align: 'center' });

 // Add chart image
 await this.#addChartImage(doc, canvas, 40);

 // Add details if available
 if (details) {
 await this.#addChartDetails(doc, details, 130);
 }

 // Add disclaimer
 await this.#addDisclaimer(doc);

 // Create filename
 const timestamp = new Date().toISOString().replace(/[-:.]/g, '').substring(0, 15);
 const filename = `cvd-risk-chart-${timestamp}.pdf`;

 // Save PDF
 doc.save(filename);

 return {

```

```
 success: true,
 filename,
 format: 'pdf'
);
} catch (error) {
 console.error('Error generating chart PDF:', error);

 return {
 success: false,
 error: error.message || 'Unknown error generating chart PDF',
 format: 'pdf'
 };
}

/**
 * Export combined view as PDF
 * @returns {Promise<Object>} - Export result
 */
static async exportCombinedView() {
 try {
 console.log('Exporting combined view as PDF...');

 // Try to import jsPDF library
 const jsPDF = (await import('jspdf')).default;
 const html2canvas = (await import('html2canvas')).default;

 // Get combined view container
 const container = document.getElementById('combined-view-container');

 if (!container) {
 throw new Error('Combined view container not found');
 }

 // Capture container as canvas
 const canvas = await html2canvas(container, {
 scale: 2,
 useCORS: true,
 logging: false
 });

 // Create new PDF document
 const doc = new jsPDF({
 orientation: 'portrait',
```

```

 unit: 'mm',
 format: 'a4'
 });

 // Add title
 const title = 'Combined Risk Assessment Report';
 doc.setFont('helvetica', 'bold');
 doc.setFontSize(18);
 doc.text(title, 105, 20, { align: 'center' });

 // Add generation date
 const dateStr = new Date().toLocaleDateString();
 doc.setFont('helvetica', 'normal');
 doc.setFontSize(10);
 doc.text(`Generated on: ${dateStr}`, 105, 27, { align: 'center' });

 // Add canvas image
 const imgData = canvas.toDataURL('image/png');
 const imgWidth = 190;
 const imgHeight = canvas.height * imgWidth / canvas.width;

 // Add image to PDF
 doc.addImage(imgData, 'PNG', 10, 35, imgWidth, imgHeight);

 // Add new page if content overflows
 if (35 + imgHeight > 270) {
 doc.addPage();
 await this.#addDisclaimer(doc);
 } else {
 await this.#addDisclaimer(doc);
 }

 // Create filename
 const timestamp = new Date().toISOString().replace(/[-.:]/g, '').substring(0, 15);
 const filename = `cvd-risk-combined-report-${timestamp}.pdf`;

 // Save PDF
 doc.save(filename);

 return {
 success: true,
 filename,
 format: 'pdf'
 };
}

```

```

 } catch (error) {
 console.error('Error exporting combined view as PDF:', error);

 return {
 success: false,
 error: error.message || 'Unknown error exporting combined view',
 format: 'pdf'
 };
 }
 }

 /**
 * Add patient information to PDF
 * @param {Object} doc - jsPDF document
 * @param {Object} data - Report data
 * @param {number} startY - Start Y position
 * @returns {number} - Next Y position
 * @private
 */
 static async #addPatientInformation(doc, data, startY) {
 try {
 // Add section title
 doc.setFont('helvetica', 'bold');
 doc.setFontSize(14);
 doc.text('Patient Information', 10, startY);

 // Extract patient data
 const patientData = data.patientData || data;

 // Add patient information table
 const tableData = [
 ['Age', patientData.age || 'N/A'],
 ['Sex', patientData.sex || 'N/A'],
 ['Blood Pressure', patientData.systolicBP ? `${patientData.systolicBP} mmHg` : 'N/A'],
 ['Total Cholesterol', patientData.totalCholesterol ? `${patientData.totalCholes}` : 'N/A'],
 ['HDL Cholesterol', patientData.hdl ? `${patientData.hdl} mmol/L` : 'N/A'],
 ['LDL Cholesterol', patientData.ldl ? `${patientData.ldl} mmol/L` : 'N/A'],
 ['Smoker', patientData.isSmoker ? 'Yes' : 'No'],
 ['Diabetes', patientData.hasDiabetes ? 'Yes' : 'No']
];

 // Add table to PDF
 doc.autoTable({
 startY: startY + 5,

```

```

 head: [],
 body: tableData,
 theme: 'plain',
 styles: {
 fontSize: 10,
 cellPadding: 2
 },
 columnStyles: {
 0: { fontStyle: 'bold', cellWidth: 40 },
 1: { cellWidth: 50 }
 },
 margin: { left: 15 }
 });

 // Return next Y position
 return doc.autoTable.previous.finalY + 10;
} catch (error) {
 console.error('Error adding patient information to PDF:', error);
 return startY + 40; // Default offset
}
}

/**
 * Add risk assessment to PDF
 * @param {Object} doc - jsPDF document
 * @param {Object} data - Report data
 * @param {number} startY - Start Y position
 * @returns {number} - Next Y position
 * @private
 */
static async #addRiskAssessment(doc, data, startY) {
 try {
 // Add section title
 doc.setFont('helvetica', 'bold');
 doc.setFontSize(14);
 doc.text('Risk Assessment Results', 10, startY);

 // Initialize table data
 const tableData = [];

 // Add FRS result if available
 if (data.frsResult) {
 const frsRisk = data.frsResult.modifiedRiskPercent || data.frsResult.tenYearRis
 const riskCategory = data.frsResult.riskCategory.charAt(0).toUpperCase() + data
 }
 }
}

```

```

 tableData.push(
 ['Framingham Risk Score', `${frsRisk}%`],
 ['Risk Category', riskCategory],
 ['', '']
);
 }

 // Add QRISK result if available
 if (data.qriskResult) {
 const riskCategory = data.qriskResult.riskCategory.charAt(0).toUpperCase() + data.qriskResult.riskCategory.slice(1);

 tableData.push(
 ['QRISK3', `${data.qriskResult.tenYearRiskPercent}%`],
 ['Risk Category', riskCategory],
 ['Heart Age', `${data.qriskResult.heartAge} years`]
);
 }

 // Add table to PDF if data available
 if (tableData.length > 0) {
 doc.autoTable({
 startY: startY + 5,
 head: [],
 body: tableData,
 theme: 'plain',
 styles: {
 fontSize: 10,
 cellPadding: 2
 },
 columnStyles: {
 0: { fontStyle: 'bold', cellWidth: 40 },
 1: { cellWidth: 50 }
 },
 margin: { left: 15 }
 });
 }

 // Return next Y position
 return doc.autoTable.previous.finalY + 10;
}

// No risk assessment data available
doc.setFont('helvetica', 'normal');
doc.setFontSize(10);

```

```

 doc.text('No risk assessment data available.', 15, startY + 10);

 return startY + 20;
 } catch (error) {
 console.error('Error adding risk assessment to PDF:', error);
 return startY + 40; // Default offset
 }
}

/**
 * Add recommendations to PDF
 * @param {Object} doc - jsPDF document
 * @param {Object} data - Report data
 * @param {number} startY - Start Y position
 * @returns {number} - Next Y position
 * @private
 */
static async #addRecommendations(doc, data, startY) {
 try {
 // Add section title
 doc.setFont('helvetica', 'bold');
 doc.setFontSize(14);
 doc.text('Treatment Recommendations', 10, startY);

 // Extract recommendations
 const recommendations = data.recommendations || {};

 // Add recommendation content
 doc.setFont('helvetica', 'normal');
 doc.setFontSize(10);

 let y = startY + 10;

 // Add LDL target
 if (recommendations.ldlTarget) {
 doc.setFont('helvetica', 'bold');
 doc.text('LDL-C Target:', 15, y);
 doc.setFont('helvetica', 'normal');
 doc.text(recommendations.ldlTarget, 50, y);
 y += 7;
 }

 // Add primary treatment
 if (recommendations.primaryTreatment) {

```

```
doc.setFont('helvetica', 'bold');
doc.text('Primary Treatment:', 15, y);
doc.setFont('helvetica', 'normal');
doc.text(recommendations.primaryTreatment, 50, y);
y += 7;
}

// Add alternative treatments
if (recommendations.alternativeTreatments) {
 doc.setFont('helvetica', 'bold');
 doc.text('Alternative Treatments:', 15, y);
 doc.setFont('helvetica', 'normal');

 const lines = doc.splitTextToSize(recommendations.alternativeTreatments, 150);
 doc.text(lines, 50, y);
 y += lines.length * 5 + 2;
}

// Add Lifestyle recommendations
if (recommendations.lifestyle && recommendations.lifestyle.length > 0) {
 doc.setFont('helvetica', 'bold');
 doc.text('Lifestyle Recommendations:', 15, y);
 doc.setFont('helvetica', 'normal');
 y += 7;

 recommendations.lifestyle.forEach(item => {
 doc.text('• ' + item, 20, y);
 y += 5;
 });
}

// Add monitoring recommendations
if (recommendations.monitoring && recommendations.monitoring.length > 0) {
 y += 2;
 doc.setFont('helvetica', 'bold');
 doc.text('Monitoring:', 15, y);
 doc.setFont('helvetica', 'normal');
 y += 7;

 recommendations.monitoring.forEach(item => {
 doc.text('• ' + item, 20, y);
 y += 5;
 });
}
```

```

 // Add guideline reference
 if (recommendations.guidelineReference) {
 y += 2;
 doc.setFontSize(8);
 doc.text(`Based on: ${recommendations.guidelineReference}` , 15, y);
 y += 5;
 }

 return y + 5;
} catch (error) {
 console.error('Error adding recommendations to PDF:', error);
 return startY + 40; // Default offset
}
}

/**
 * Add disclaimer to PDF
 * @param {Object} doc - jsPDF document
 * @returns {void}
 * @private
 */
static async #addDisclaimer(doc) {
 try {
 // Add disclaimer at bottom of page
 const disclaimer = 'DISCLAIMER: This report is intended for educational and informa';

 const pageHeight = doc.internal.pageSize.height;
 const pageWidth = doc.internal.pageSize.width;

 doc.setFont('helvetica', 'normal');
 doc.setFontSize(8);

 const textWidth = pageWidth - 20;
 const lines = doc.splitTextToSize(disclaimer, textWidth);
 const textHeight = lines.length * 3.5;

 // Draw disclaimer box
 doc.setDrawColor(0);
 doc.setFillColor(240, 240, 240);
 doc.roundedRect(10, pageHeight - textHeight - 10, pageWidth - 20, textHeight, 2, 2);

 // Add disclaimer text
 doc.text(lines, 15, pageHeight - textHeight - 5);
 }
}

```

```

 } catch (error) {
 console.error('Error adding disclaimer to PDF:', error);
 }
}

/**
 * Add chart image to PDF
 * @param {Object} doc - jsPDF document
 * @param {HTMLCanvasElement} canvas - Chart canvas
 * @param {number} startY - Start Y position
 * @returns {number} - Next Y position
 * @private
 */
static async #addChartImage(doc, canvas, startY) {
 try {
 // Convert canvas to image data
 const imgData = canvas.toDataURL('image/png');

 // Calculate dimensions
 const pageWidth = doc.internal.pageSize.width;
 const imgWidth = pageWidth - 20;
 const imgHeight = canvas.height * imgWidth / canvas.width;

 // Add image to PDF
 doc.addImage(imgData, 'PNG', 10, startY, imgWidth, imgHeight);

 return startY + imgHeight + 10;
 } catch (error) {
 console.error('Error adding chart image to PDF:', error);
 return startY + 80; // Default offset
 }
}

/**
 * Add chart details to PDF
 * @param {Object} doc - jsPDF document
 * @param {string} details - Chart details HTML
 * @param {number} startY - Start Y position
 * @returns {number} - Next Y position
 * @private
 */
static async #addChartDetails(doc, details, startY) {
 try {
 // Parse HTML details

```

```

 const parser = new DOMParser();
 const detailsDoc = parser.parseFromString(details, 'text/html');

 // Extract text content
 const textContent = detailsDoc.body.textContent.trim();

 // Skip if no content
 if (!textContent) {
 return startY;
 }

 // Add section title
 doc.setFont('helvetica', 'bold');
 doc.setFontSize(12);
 doc.text('Chart Details', 10, startY);

 // Add details content
 doc.setFont('helvetica', 'normal');
 doc.setFontSize(10);

 const pageWidth = doc.internal.pageSize.width;
 const textWidth = pageWidth - 20;
 const lines = doc.splitTextToSize(textContent, textWidth);

 doc.text(lines, 10, startY + 8);

 return startY + lines.length * 5 + 10;
} catch (error) {
 console.error('Error adding chart details to PDF:', error);
 return startY + 20; // Default offset
}

/**
 * Add charts to PDF
 * @param {Object} doc - jsPDF document
 * @param {Array<Object>} charts - Charts data
 * @returns {void}
 * @private
 */
static async #addCharts(doc, charts) {
 try {
 // Add new page for charts
 doc.addPage();

```

```

// Add charts title
doc.setFont('helvetica', 'bold');
doc.setFontSize(16);
doc.text('Risk Visualizations', 105, 20, { align: 'center' });

let y = 30;

// Process each chart
for (const chart of charts) {
 // Check if need to add new page
 if (y > 200) {
 doc.addPage();
 y = 20;
 }

 // Add chart title
 doc.setFont('helvetica', 'bold');
 doc.setFontSize(12);
 doc.text(chart.title || 'Chart', 10, y);

 // Add chart image
 if (chart.canvas) {
 const imgData = chart.canvas.toDataURL('image/png');
 const imgWidth = 180;
 const imgHeight = chart.canvas.height * imgWidth / chart.canvas.width;

 // Add image to PDF
 doc.addImage() // Try to determine type
 const diabetesType = cond.code.coding.find(c =>
 c.code === '46635009' || c.code === '44054006'
);

 if (diabetesType) {
 data.diabetesStatus = diabetesType.code === '46635009' ? 'type1' : 'typ
 } else {
 data.diabetesStatus = 'type2'; // Default to type 2
 }
 break;
 }

 case '22298006': // Atrial fibrillation
 data.atrialFibrillation = true;
 break;
}

```

```

 case '73211009': // Rheumatoid arthritis
 data.rheumatoidArthritis = true;
 break;

 case '396332003': // Chronic kidney disease
 data.chronicKidneyDisease = true;
 break;

 case '56265001': // Heart disease
 case '53741008': // Coronary arteriosclerosis
 data.establishedASCVD = true;
 break;
 }
}

return data;
}

/**
 * Map patient data to FHIR resources
 * @param {Object} data - Patient data
 * @returns {Object} - FHIR resources
 * @private
 */
static #mapPatientDataToFHIR(data) {
 const fhirData = {
 patient: {
 resourceType: 'Patient',
 id: data.patientId || `patient-${new Date().getTime()}`,
 meta: {
 profile: ['http://hl7.org/fhir/StructureDefinition/Patient']
 },
 active: true,
 gender: data.sex || 'unknown',
 extension: []
 },
 observations: [],
 conditions: [],
 riskAssessment: null
 };

 // Add smoking status extension
 if (data.isSmoker !== undefined) {
 fhirData.patient.extension.push({

```

```

 url: 'http://hl7.org/fhir/StructureDefinition/patient-smokingStatus',
 valueCodeableConcept: {
 coding: [
 {
 system: 'http://snomed.info/sct',
 code: data.isSmoker ? '449868002' : '266919005',
 display: data.isSmoker ? 'Current smoker' : 'Never smoked'
 }
],
 text: data.isSmoker ? 'Current smoker' : 'Never smoked'
 }
}

// Calculate birth date from age (approximate)
if (data.age) {
 const birthYear = new Date().getFullYear() - data.age;
 fhirData.patient.birthDate = `${birthYear}-01-01`;
}

// Add observations
const obsDate = new Date().toISOString();

// Systolic BP
if (data.systolicBP) {
 fhirData.observations.push({
 resourceType: 'Observation',
 id: `obs-sbp-${new Date().getTime()}`,
 status: 'final',
 category: [
 {
 coding: [
 {
 system: 'http://terminology.hl7.org/CodeSystem/observation-category',
 code: 'vital-signs',
 display: 'Vital Signs'
 }
]
 }
],
 code: {
 coding: [
 {
 system: 'http://loinc.org',

```

```

 code: '8480-6',
 display: 'Systolic blood pressure'
 }
],
text: 'Systolic blood pressure'
},
subject: {
 reference: `Patient/${fhirData.patient.id}`
},
effectiveDateTime: obsDate,
valueQuantity: {
 value: data.systolicBP,
 unit: 'mmHg',
 system: 'http://unitsofmeasure.org',
 code: 'mm[Hg]'
}
});
}

// Total cholesterol
if (data.totalCholesterol) {
 fhirData.observations.push({
 resourceType: 'Observation',
 id: `obs-tc-${new Date().getTime()}`,
 status: 'final',
 category: [
 {
 coding: [
 {
 system: 'http://terminology.hl7.org/CodeSystem/observation-category',
 code: 'laboratory',
 display: 'Laboratory'
 }
]
 }
],
 code: {
 coding: [
 {
 system: 'http://loinc.org',
 code: '14647-2',
 display: 'Total cholesterol'
 }
],
 }
 });
}

```

```

 text: 'Total cholesterol'
 },
 subject: {
 reference: `Patient/${fhirData.patient.id}`
 },
 effectiveDateTime: obsDate,
 valueQuantity: {
 value: data.totalCholesterol,
 unit: 'mmol/L',
 system: 'http://unitsofmeasure.org',
 code: 'mmol/L'
 }
});

}

// HDL
if (data.hdl) {
 fhirData.observations.push({
 resourceType: 'Observation',
 id: `obs-hdl-${new Date().getTime()}`,
 status: 'final',
 category: [
 {
 coding: [
 {
 system: 'http://terminology.hl7.org/CodeSystem/observation-category',
 code: 'laboratory',
 display: 'Laboratory'
 }
]
 }
],
 code: {
 coding: [
 {
 system: 'http://loinc.org',
 code: '14646-4',
 display: 'HDL cholesterol'
 }
],
 text: 'HDL cholesterol'
 },
 subject: {
 reference: `Patient/${fhirData.patient.id}`
 }
 })
}

```

```

 },
 effectiveDateTime: obsDate,
 valueQuantity: {
 value: data.hdl,
 unit: 'mmol/L',
 system: 'http://unitsofmeasure.org',
 code: 'mmol/L'
 }
});

}

// Add conditions
const recordedDate = new Date().toISOString();

// Diabetes
if (data.hasDiabetes) {
 const diabetesCode = data.diabetesStatus === 'type1' ? '46635009' : '44054006';
 const diabetesDisplay = data.diabetesStatus === 'type1' ? 'Type 1 diabetes mellitus'

 fhirData.conditions.push({
 resourceType: 'Condition',
 id: `cond-diabetes-${new Date().getTime()}`,
 clinicalStatus: {
 coding: [
 {
 system: 'http://terminology.hl7.org/CodeSystem/condition-clinical',
 code: 'active',
 display: 'Active'
 }
]
 },
 verificationStatus: {
 coding: [
 {
 system: 'http://terminology.hl7.org/CodeSystem/condition-ver-status',
 code: 'confirmed',
 display: 'Confirmed'
 }
]
 },
 category: [
 {
 coding: [
 {

```

```

 system: 'http://terminology.hl7.org/CodeSystem/condition-categc
 code: 'problem-list-item',
 display: 'Problem List Item'
 }
]
}
],
code: {
 coding: [
 {
 system: 'http://snomed.info/sct',
 code: diabetesCode,
 display: diabetesDisplay
 }
],
 text: diabetesDisplay
},
subject: {
 reference: `Patient/${fhirData.patient.id}`
},
recordedDate: recordedDate
});
}

// Create risk assessment if data available
if (data.frsResult || data.qriskResult) {
 const riskResult = data.frsResult || data.qriskResult;
 const calculator = data.frsResult ? 'framingham' : 'qrisk3';

 fhirData.riskAssessment = {
 resourceType: 'RiskAssessment',
 id: `risk-${calculator}-${new Date().getTime()}`,
 status: 'final',
 code: {
 coding: [
 {
 system: 'http://snomed.info/sct',
 code: calculator === 'framingham' ? '441829007' : '871186003',
 display: calculator === 'framingham' ? 'Framingham cardiovascular c
 }
],
 text: calculator === 'framingham' ? 'Framingham Risk Score' : 'QRISK3'
 },
 subject: {

```

```

 reference: `Patient/${fhirData.patient.id}`
 },
 occurrenceDateTime: riskResult.calculationDate || new Date().toISOString(),
 prediction: [
 {
 outcome: {
 coding: [
 {
 system: 'http://snomed.info/sct',
 code: '441829007',
 display: 'Cardiovascular disease'
 }
],
 text: 'Cardiovascular disease'
 },
 probabilityDecimal: (riskResult.modifiedRiskPercent || riskResult.tenYearRisk) / 100
 },
 whenRange: {
 low: {
 value: 0,
 unit: 'years',
 system: 'http://unitsofmeasure.org',
 code: 'a'
 },
 high: {
 value: 10,
 unit: 'years',
 system: 'http://unitsofmeasure.org',
 code: 'a'
 }
 }
]
};

return fhirData;
}

/**
 * Send FHIR API request
 * @param {string} path - API path
 * @param {Object} options - Request options
 * @returns {Promise<Object>} - Response data
 * @private

```

```

/*
static async #sendRequest(path, options = {}) {
 try {
 // Build URL
 const url = new URL(path, this.#endpoint);

 // Set default headers
 const headers = options.headers || {};
 headers['Accept'] = 'application/fhir+json';

 if (options.method !== 'GET') {
 headers['Content-Type'] = 'application/fhir+json';
 }

 // Add authorization if available
 if (this.#authType === 'bearer' && this.#authToken) {
 headers['Authorization'] = `Bearer ${this.#authToken}`;
 } else if (this.#authType === 'basic' && this.#authToken) {
 headers['Authorization'] = `Basic ${this.#authToken}`;
 }

 // Send request
 const response = await fetch(url.toString(), {
 ...options,
 headers
 });

 // Check if successful
 if (!response.ok) {
 throw new Error(`FHIR API request failed: ${response.status} ${response.statusText}`);
 }

 // Parse JSON response
 return await response.json();
 } catch (error) {
 console.error('Error sending FHIR API request:', error);
 throw error;
 }
}

/**
 * Get patient name
 * @param {Object} patient - FHIR Patient resource
 * @returns {string} - Patient name

```

```

* @private
*/
static #getPatientName(patient) {
 if (!patient.name || !patient.name.length) {
 return 'Unknown';
 }

 const name = patient.name[0];
 const given = name.given ? name.given.join(' ') : '';
 const family = name.family || '';

 return `${given} ${family}`.trim();
}

/**
 * Calculate age from birth date
 * @param {string} birthDate - Birth date in YYYY-MM-DD format
 * @returns {number} - Age in years
 * @private
*/
static #calculateAgeFromBirthDate(birthDate) {
 if (!birthDate) {
 return null;
 }

 try {
 const dob = new Date(birthDate);
 const today = new Date();

 let age = today.getFullYear() - dob.getFullYear();
 const monthDiff = today.getMonth() - dob.getMonth();

 if (monthDiff < 0 || (monthDiff === 0 && today.getDate() < dob.getDate())) {
 age--;
 }

 return age;
 } catch (error) {
 console.error('Error calculating age from birth date:', error);
 return null;
 }
}

```

```
export default EMRConnector;
```

## 4.3 field-mapper.js

javascript

```

/**
 * Field Mapper for CVD Risk Toolkit
 *
 * Maps between different naming conventions and data models,
 * supporting import/export with different systems.
 */

class FieldMapper {
 // Default field mappings
 static #defaultMappings = {
 // Common field mappings
 common: {
 // Demographics
 'age': ['age', 'Age', 'AGE', 'PatientAge'],
 'sex': ['sex', 'gender', 'Gender', 'Sex', 'SEX', 'PatientGender'],
 'ethnicity': ['ethnicity', 'ethnic_origin', 'EthnicOrigin', 'Ethnicity'],
 'height': ['height', 'Height', 'HeightCm', 'height_cm'],
 'weight': ['weight', 'Weight', 'WeightKg', 'weight_kg'],
 'bmi': ['bmi', 'BMI', 'BodyMassIndex'],
 }

 // Vitals
 'systolicBP': ['systolicBP', 'sbp', 'SBP', 'SystolicBP', 'systolic', 'SystolicBlooc
 'diastolicBP': ['diastolicBP', 'dbp', 'DBP', 'DiastolicBP', 'diastolic', 'Diastolic
 'onBPMeds': ['onBPMeds', 'bp_meds', 'onHypertensionMeds', 'onHTNMed', 'onBPMedicat

 // Lipids
 'totalCholesterol': ['totalCholesterol', 'total_cholesterol', 'tc', 'TC', 'TotalChc
 'hdl': ['hdl', 'HDL', 'hdl_cholesterol', 'HDLCholesterol'],
 'ldl': ['ldl', 'LDL', 'ldl_cholesterol', 'LDLCholesterol'],
 'triglycerides': ['triglycerides', 'tg', 'TG', 'Triglycerides'],
 'cholesterolRatio': ['cholesterolRatio', 'tc_hdl_ratio', 'TCHDLRatio', 'totalHDLRat

 // Risk factors
 'isSmoker': ['isSmoker', 'smoker', 'Smoker', 'smokingStatus', 'current_smoker'],
 'smokingStatus': ['smokingStatus', 'smoking_status', 'SmokingCategory'],
 'hasDiabetes': ['hasDiabetes', 'diabetes', 'Diabetes', 'diabetic'],
 'diabetesStatus': ['diabetesStatus', 'diabetes_type', 'DiabetesType'],
 'familyHistory': ['familyHistory', 'family_history', 'FamilyHistoryCHD', 'familyHis
 'atrialFibrillation': ['atrialFibrillation', 'af', 'AF', 'AtrialFibrillation'],
 'rheumatoidArthritis': ['rheumatoidArthritis', 'ra', 'RA', 'RheumatoidArthritis'],
 'chronicKidneyDisease': ['chronicKidneyDisease', 'ckd', 'CKD', 'ChronicKidneyDiseas
 'establishedASCVD': ['establishedASCVD', 'ascvd', 'ASCVD', 'EstablishedCVD']

 },
}

```

```

// EPIC EMR field mappings
epic: {
 'age': ['Age', 'AGE', 'PAT_AGE_YRS'],
 'sex': ['Gender', 'SEX', 'PAT_GENDER'],
 'heightCm': ['Height', 'PAT_HT_CM'],
 'weightKg': ['Weight', 'PAT_WT_KG'],
 'systolicBP': ['SBP', 'PAT_SBP_SYSTOLIC'],
 'diastolicBP': ['DBP', 'PAT_DBP_DIASTOLIC'],
 'totalCholesterol': ['Total_Chol', 'PAT_TOT_CHOL'],
 'hdl': ['HDL', 'PAT_HDL'],
 'ldl': ['LDL', 'PAT_LDL'],
 'isSmoker': ['Smoker', 'PAT_SMOKER_YN'],
 'hasDiabetes': ['Diabetes', 'PAT_DIABETES_YN']
},

// Cerner EMR field mappings
cerner: {
 'age': ['Age', 'AGE_YEARS'],
 'sex': ['Gender', 'SEX'],
 'heightCm': ['Height', 'HEIGHT_CM'],
 'weightKg': ['Weight', 'WEIGHT_KG'],
 'systolicBP': ['SystolicBP', 'SYSTOLIC_BP'],
 'diastolicBP': ['DiastolicBP', 'DIASTOLIC_BP'],
 'totalCholesterol': ['TotalCholesterol', 'TOT_CHOL'],
 'hdl': ['HDL', 'HDL_CHOL'],
 'ldl': ['LDL', 'LDL_CHOL'],
 'isSmoker': ['CurrentSmoker', 'IS_SMOKER'],
 'hasDiabetes': ['Diabetes', 'HAS_DIABETES']
},

// QRISK3 field mappings
qrisk3: {
 'age': ['age'],
 'sex': ['sex'],
 'ethnicity': ['ethnicity'],
 'systolicBP': ['systolicBP', 'sbp'],
 'systolicBP_sd': ['systolicBP_sd', 'sbp_sd'],
 'cholesterolRatio': ['cholesterolRatio', 'cholhdl'],
 'height': ['height'],
 'weight': ['weight'],
 'bmi': ['bmi'],
 'smokingStatus': ['smokingStatus'],
 'diabetesStatus': ['diabetesStatus', 'diabetesType'],
}

```

```

 'familyHistory': ['familyHistory', 'fh_cvd'],
 'atrialFibrillation': ['atrialFibrillation', 'af'],
 'onBPMeds': ['onBPMeds', 'treatedhyp'],
 'rheumatoidArthritis': ['rheumatoidArthritis', 'ra'],
 'chronicKidneyDisease': ['chronicKidneyDisease', 'ckd'],
 'migraines': ['migraines'],
 'erectileDysfunction': ['erectileDysfunction', 'ed']
 },
}

// Framingham field mappings
framingham: {
 'age': ['age'],
 'sex': ['sex'],
 'systolicBP': ['systolicBP', 'sbp'],
 'onBPMeds': ['onBPMeds', 'BPTreatment'],
 'totalCholesterol': ['totalCholesterol', 'totalChol'],
 'hdl': ['hdl', 'hdlc'],
 'ldl': ['ldl', 'ldlc'],
 'isSmoker': ['isSmoker', 'smoker'],
 'hasDiabetes': ['hasDiabetes', 'diabetes'],
 'familyHistory': ['familyHistory', 'FHx_HeartDisease'],
 'lpa': ['lpa', 'lipoproteinA'],
 'isSouthAsian': ['isSouthAsian', 'southAsian']
}
};

// Custom mappings
static #customMappings = new Map();

/**
 * Get field mappings
 * @param {string} system - System name
 * @returns {Object} - Field mappings
 */
static getMappings(system) {
 // Get custom mappings if available
 if (this.#customMappings.has(system)) {
 return this.#customMappings.get(system);
 }

 // Get default mappings
 return this.#defaultMappings[system] || this.#defaultMappings.common;
}

```

```

/**
 * Set custom field mappings
 * @param {string} system - System name
 * @param {Object} mappings - Field mappings
 */
static setMappings(system, mappings) {
 this.#customMappings.set(system, mappings);
}

/**
 * Map field name
 * @param {string} fieldName - Field name to map
 * @param {string} fromSystem - Source system
 * @param {string} toSystem - Target system
 * @returns {string} - Mapped field name
*/
static mapFieldName(fieldName, fromSystem, toSystem) {
 try {
 // Get source system mappings
 const sourceMappings = this.getMappings(fromSystem);

 // Find canonical field name
 let canonicalField = null;

 for (const [field, aliases] of Object.entries(sourceMappings)) {
 if (field === fieldName || aliases.includes(fieldName)) {
 canonicalField = field;
 break;
 }
 }

 if (!canonicalField) {
 // If no mapping found, return original field name
 return fieldName;
 }

 // Get target system mappings
 const targetMappings = this.getMappings(toSystem);

 // Find target field name
 for (const [field, aliases] of Object.entries(targetMappings)) {
 if (field === canonicalField || aliases.includes(canonicalField)) {
 return field; // Return the primary field name for the target system
 }
 }
 }
}

```

```

 }

 // If no mapping found, return canonical field name
 return canonicalField;
 } catch (error) {
 console.error('Error mapping field name:', error);
 return fieldName; // Return original field name on error
 }
}

/**
 * Map field value
 * @param {string} fieldName - Field name
 * @param {*} value - Field value
 * @param {string} fromSystem - Source system
 * @param {string} toSystem - Target system
 * @returns {*} - Mapped field value
 */
static mapFieldValue(fieldName, value, fromSystem, toSystem) {
 try {
 // Skip if value is null or undefined
 if (value === null || value === undefined) {
 return value;
 }

 // Map boolean fields
 if (['isSmoker', 'hasDiabetes', 'onBPMeds', 'familyHistory',
 'atrialFibrillation', 'rheumatoidArthritis', 'chronicKidneyDisease',
 'establishedASCVD', 'erectileDysfunction'].includes(fieldName)) {
 return this.#mapBooleanValue(value);
 }

 // Map sex field
 if (fieldName === 'sex' || fieldName === 'gender') {
 return this.#mapSexValue(value);
 }

 // Map smoking status
 if (fieldName === 'smokingStatus') {
 return this.#mapSmokingStatusValue(value, fromSystem, toSystem);
 }

 // Map diabetes status
 if (fieldName === 'diabetesStatus') {

```

```

 return this.#mapDiabetesStatusValue(value, fromSystem, toSystem);
 }

 // Map ethnicity
 if (fieldName === 'ethnicity') {
 return this.#mapEthnicityValue(value, fromSystem, toSystem);
 }

 // Map unit conversions if needed
 if (['height', 'weight', 'totalCholesterol', 'hdl', 'ldl', 'triglycerides'].includes(fieldName)) {
 return this.#mapUnitConversion(fieldName, value, fromSystem, toSystem);
 }

 // Return original value for other fields
 return value;
} catch (error) {
 console.error('Error mapping field value:', error);
 return value; // Return original value on error
}
}

/**
 * Map entire data object
 * @param {Object} data - Data object to map
 * @param {string} fromSystem - Source system
 * @param {string} toSystem - Target system
 * @returns {Object} - Mapped data object
 */
static mapData(data, fromSystem, toSystem) {
 try {
 const mappedData = {};

 // Process each field in the data object
 for (const [fieldName, value] of Object.entries(data)) {
 // Map field name
 const mappedFieldName = this.mapFieldName(fieldName, fromSystem, toSystem);

 // Map field value
 const mappedValue = this.mapFieldValue(mappedFieldName, value, fromSystem, toSystem);

 // Add to mapped data
 mappedData[mappedFieldName] = mappedValue;
 }
 }
}
```

```

 return mappedData;
 } catch (error) {
 console.error('Error mapping data:', error);
 return data; // Return original data on error
 }
}

/***
 * Map boolean value
 * @param {*} value - Value to map
 * @returns {boolean} - Mapped boolean value
 * @private
 */
static #mapBooleanValue(value) {
 if (typeof value === 'boolean') {
 return value;
 } else if (typeof value === 'string') {
 const lowerValue = value.toLowerCase();
 return lowerValue === 'true' || lowerValue === 'yes' || lowerValue === 'y' || lowerValue === 'on';
 } else if (typeof value === 'number') {
 return value !== 0;
 } else {
 return Boolean(value);
 }
}

/***
 * Map sex value
 * @param {*} value - Value to map
 * @returns {string} - Mapped sex value
 * @private
 */
static #mapSexValue(value) {
 if (!value) return null;

 if (typeof value === 'string') {
 const lowerValue = value.toLowerCase();

 if (lowerValue === 'm' || lowerValue === 'male' || lowerValue === '1') {
 return 'male';
 } else if (lowerValue === 'f' || lowerValue === 'female' || lowerValue === '2') {
 return 'female';
 }
 }
}

```

```

 return value;
}

/**
 * Map smoking status value
 * @param {*} value - Value to map
 * @param {string} fromSystem - Source system
 * @param {string} toSystem - Target system
 * @returns {string} - Mapped smoking status value
 * @private
*/
static #mapSmokingStatusValue(value, fromSystem, toSystem) {
 if (!value) return null;

 // Define mapping tables
 const commonMapping = {
 // Common smoking status values
 'non': ['non', 'non-smoker', 'never', 'never-smoker', 'no', '0', 'n', 'never_smokec',
 'ex': ['ex', 'ex-smoker', 'former', 'former-smoker', 'quit', 'previous'],
 'light': ['light', 'light-smoker', 'occasional', 'social', '1-9'],
 'moderate': ['moderate', 'moderate-smoker', 'regular', '10-19'],
 'heavy': ['heavy', 'heavy-smoker', 'severe', '20+']
 };

 // System-specific mappings
 const systemMappings = {
 epic: {
 'non': ['0', 'N', 'Never'],
 'ex': ['3', 'Former'],
 'light': ['1', 'Y', 'Current'],
 'moderate': ['1', 'Y', 'Current'],
 'heavy': ['1', 'Y', 'Current']
 },
 cerner: {
 'non': ['NON_SMOKER', 'NEVER', '0'],
 'ex': ['EX_SMOKER', 'FORMER', '2'],
 'light': ['CURRENT_LIGHT', '1'],
 'moderate': ['CURRENT_MOD', '1'],
 'heavy': ['CURRENT_HEAVY', '1']
 }
 };
}

// Convert to lowercase if string

```

```

const normalizedValue = typeof value === 'string' ? value.toLowerCase() : value;

// Try to find match in system-specific mapping
if (fromSystem in systemMappings) {
 for (const [status, values] of Object.entries(systemMappings[fromSystem])) {
 if (values.map(v => v.toLowerCase()).includes(normalizedValue)) {
 return status;
 }
 }
}

// Try to find match in common mapping
for (const [status, values] of Object.entries(commonMapping)) {
 if (values.includes(normalizedValue)) {
 return status;
 }
}

// If value is exactly one of the status codes, return it
if (['non', 'ex', 'light'] {
 resourceType: 'Observation',
 id: `obs-ldl-${new Date().getTime()}`,
 status: 'final',
 category: [
 {
 coding: [
 {
 system: 'http://terminology.hl7.org/CodeSystem/observation-category',
 code: 'laboratory',
 display: 'Laboratory'
 }
]
 }
],
 code: {
 coding: [
 {
 system: 'http://loinc.org',
 code: '13457-7',
 display: 'LDL cholesterol'
 }
],
 text: 'LDL cholesterol'
 },
 subject: {

```

```

 reference: `Patient/${patientId}`
 },
 effectiveDateTime: obsDate,
 valueQuantity: {
 value: data.ldl,
 unit: 'mmol/L',
 system: 'http://unitsofmeasure.org',
 code: 'mmol/L'
 }
}
});

}

// Triglycerides
if (data.triglycerides) {
 observations.push({
 resourceType: 'Observation',
 id: `obs-trig-${new Date().getTime()}`,
 status: 'final',
 category: [
 {
 coding: [
 {
 system: 'http://terminology.hl7.org/CodeSystem/observation-category',
 code: 'laboratory',
 display: 'Laboratory'
 }
]
 }
],
 code: {
 coding: [
 {
 system: 'http://loinc.org',
 code: '14927-8',
 display: 'Triglycerides'
 }
],
 text: 'Triglycerides'
 },
 subject: {
 reference: `Patient/${patientId}`
 },
 effectiveDateTime: obsDate,
 valueQuantity: {

```

```

 value: data.triglycerides,
 unit: 'mmol/L',
 system: 'http://unitsofmeasure.org',
 code: 'mmol/L'
 }
});

}

return observations;
}

/**
 * Create FHIR Condition resources
 * @param {Object} data - Patient data
 * @returns {Array<Object>} - FHIR Condition resources
 * @private
 */
static #createFHIRConditionResources(data) {
 const conditions = [];
 const patientId = `patient-${new Date().getTime()}`;
 const recordedDate = new Date().toISOString();

 // Diabetes
 if (data.hasDiabetes) {
 conditions.push({
 resourceType: 'Condition',
 id: `cond-diabetes-${new Date().getTime()}`,
 clinicalStatus: {
 coding: [
 {
 system: 'http://terminology.hl7.org/CodeSystem/condition-clinical',
 code: 'active',
 display: 'Active'
 }
]
 },
 verificationStatus: {
 coding: [
 {
 system: 'http://terminology.hl7.org/CodeSystem/condition-ver-status',
 code: 'confirmed',
 display: 'Confirmed'
 }
]
 }
 });
 }
}

```

```
 },
 category: [
 {
 coding: [
 {
 system: 'http://terminology.hl7.org/CodeSystem/condition-category',
 code: 'problem-list-item',
 display: 'Problem List Item'
 }
]
 }
],
 code: {
 coding: [
 {
 system: 'http://snomed.info/sct',
 code: '44054006',
 display: 'Diabetes mellitus'
 }
],
 text: 'Diabetes mellitus'
 },
 subject: {
 reference: `Patient/${patientId}`
 },
 recordedDate: recordedDate
 });
}

// Atrial fibrillation
if (data.atrialFibrillation) {
 conditions.push({
 resourceType: 'Condition',
 id: `cond-af-${new Date().getTime()}`,
 clinicalStatus: {
 coding: [
 {
 system: 'http://terminology.hl7.org/CodeSystem/condition-clinical',
 code: 'active',
 display: 'Active'
 }
]
 },
 verificationStatus: {

```

```

 coding: [
 {
 system: 'http://terminology.hl7.org/CodeSystem/condition-ver-status',
 code: 'confirmed',
 display: 'Confirmed'
 }
]
 },
 category: [
 {
 coding: [
 {
 system: 'http://terminology.hl7.org/CodeSystem/condition-category',
 code: 'problem-list-item',
 display: 'Problem List Item'
 }
]
 }
],
 code: {
 coding: [
 {
 system: 'http://snomed.info/sct',
 code: '22298006',
 display: 'Atrial fibrillation'
 }
],
 text: 'Atrial fibrillation'
 },
 subject: {
 reference: `Patient/${patientId}`
 },
 recordedDate: recordedDate
});

}

// Rheumatoid arthritis
if (data.rheumatoidArthritis) {
 conditions.push({
 resourceType: 'Condition',
 id: `cond-ra-${new Date().getTime()}`,
 clinicalStatus: {
 coding: [
 {

```

```
 system: 'http://terminology.hl7.org/CodeSystem/condition-clinical',
 code: 'active',
 display: 'Active'
 }
]
},
verificationStatus: {
 coding: [
 {
 system: 'http://terminology.hl7.org/CodeSystem/condition-ver-status',
 code: 'confirmed',
 display: 'Confirmed'
 }
]
},
category: [
{
 coding: [
 {
 system: 'http://terminology.hl7.org/CodeSystem/condition-category',
 code: 'problem-list-item',
 display: 'Problem List Item'
 }
]
}
],
code: {
 coding: [
 {
 system: 'http://snomed.info/sct',
 code: '73211009',
 display: 'Rheumatoid arthritis'
 }
],
 text: 'Rheumatoid arthritis'
},
subject: {
 reference: `Patient/${patientId}`
},
recordedDate: recordedDate
});
}

// Chronic kidney disease
```

```
if (data.chronicKidneyDisease) {
 conditions.push({
 resourceType: 'Condition',
 id: `cond-ckd-${new Date().getTime()}`,
 clinicalStatus: {
 coding: [
 {
 system: 'http://terminology.hl7.org/CodeSystem/condition-clinical',
 code: 'active',
 display: 'Active'
 }
]
 },
 verificationStatus: {
 coding: [
 {
 system: 'http://terminology.hl7.org/CodeSystem/condition-ver-status',
 code: 'confirmed',
 display: 'Confirmed'
 }
]
 },
 category: [
 {
 coding: [
 {
 system: 'http://terminology.hl7.org/CodeSystem/condition-category',
 code: 'problem-list-item',
 display: 'Problem List Item'
 }
]
 }
],
 code: {
 coding: [
 {
 system: 'http://snomed.info/sct',
 code: '396332003',
 display: 'Chronic kidney disease'
 }
],
 text: 'Chronic kidney disease'
 },
 subject: {
```

```

 reference: `Patient/${patientId}`
 },
 recordedDate: recordedDate
});

}

return conditions;
}

/**
 * Create FHIR RiskAssessment resource
 * @param {Object} riskResult - Risk assessment result
 * @param {string} calculator - Calculator type
 * @returns {Object} - FHIR RiskAssessment resource
 * @private
 */
static #createFHIRRiskAssessmentResource(riskResult, calculator) {
 const patientId = `patient-${new Date().getTime()}`;
 const riskDate = riskResult.calculationDate ? new Date(riskResult.calculationDate).toISOString();

 return {
 resourceType: 'RiskAssessment',
 id: `risk-${calculator}-${new Date().getTime()}`,
 status: 'final',
 code: {
 coding: [
 {
 system: 'http://snomed.info/sct',
 code: calculator === 'framingham' ? '441829007' : '871186003',
 display: calculator === 'framingham' ? 'Framingham cardiovascular disease risk score' : 'QRISK3'
 }
],
 text: calculator === 'framingham' ? 'Framingham Risk Score' : 'QRISK3'
 },
 subject: {
 reference: `Patient/${patientId}`
 },
 occurrenceDateTime: riskDate,
 prediction: [
 {
 outcome: {
 coding: [
 {
 system: 'http://snomed.info/sct',
 code: calculator === 'framingham' ? '441829007' : '871186003',
 display: calculator === 'framingham' ? 'Framingham cardiovascular disease risk score' : 'QRISK3'
 }
]
 }
 }
]
 }
}

```

```

 code: '441829007',
 display: 'Cardiovascular disease'
 }
],
text: 'Cardiovascular disease'

},
probabilityDecimal: (riskResult.modifiedRiskPercent || riskResult.tenYearRisk)
whenRange: {
 low: {
 value: 0,
 unit: 'years',
 system: 'http://unitsofmeasure.org',
 code: 'a'
 },
 high: {
 value: 10,
 unit: 'years',
 system: 'http://unitsofmeasure.org',
 code: 'a'
 }
}
}
]
};

}

}

/**
 * Flatten object for CSV export
 * @param {Object} obj - Object to flatten
 * @param {string} prefix - Prefix for keys
 * @returns {Object} - Flattened object
 * @private
 */
static #flattenObjectForCSV(obj, prefix = '') {
 const result = {};

 // Process patientData directly
 const patientData = obj.patientData || obj;

 for (const key in patientData) {
 const value = patientData[key];

 if (value !== null && typeof value === 'object' && !Array.isArray(value)) {
 // Flatten nested object
 result[prefix + key] = value;
 } else {
 result[prefix + key] = value;
 }
 }

 return result;
}

```

```

 const nested = this.#flattenObjectForCSV(value, `${prefix}${key}.`);
 Object.assign(result, nested);
 } else {
 // Add value to result
 result[`${prefix}${key}`] = value;
 }
}

// Add basic risk results
if (obj.frsResult) {
 result['frs.tenYearRisk'] = obj.frsResult.tenYearRiskPercent;
 result['frs.modifiedRisk'] = obj.frsResult.modifiedRiskPercent || obj.frsResult.ter
 result['frs.riskCategory'] = obj.frsResult.riskCategory;
 result['frs.calculationDate'] = obj.frsResult.calculationDate;
}

if (obj.qriskResult) {
 result['qrisk.tenYearRisk'] = obj.qriskResult.tenYearRiskPercent;
 result['qrisk.riskCategory'] = obj.qriskResult.riskCategory;
 result['qrisk.heartAge'] = obj.qriskResult.heartAge;
 result['qrisk.calculationDate'] = obj.qriskResult.calculationDate;
}

return result;
}

/**
 * Download file
 * @param {string} url - File URL
 * @param {string} filename - File name
 * @private
 */
static #downloadFile(url, filename) {
 // Create a temporary link
 const link = document.createElement('a');

 // Set link properties
 link.href = url;
 link.download = filename;

 // Click link to trigger download
 document.body.appendChild(link);
 link.click();
 document.body.removeChild(link);
}

```

```

 // Release URL object
 setTimeout(() => {
 URL.revokeObjectURL(url);
 }, 100);
}

/**
 * Populate form with data
 * @param {HTMLFormElement} form - Form to populate
 * @param {Object} data - Data to populate form with
 * @private
 */
static #populateForm(form, data) {
 if (!form || !data) return;

 // Process each form element
 const elements = Array.from(form.elements);

 elements.forEach(element => {
 const name = element.name;
 if (!name) return;

 const value = data[name];
 if (value === undefined) return;

 // Handle different element types
 if (element.type === 'checkbox') {
 element.checked = Boolean(value);
 } else if (element.type === 'radio') {
 element.checked = element.value === String(value);
 } else {
 element.value = value;
 }

 // Dispatch change event to trigger calculations
 element.dispatchEvent(new Event('change', { bubbles: true }));
 });
}

export default DataImportExport;

```

## **4.2 emr-connector.js**

javascript

```
/**
 * EMR Connector for CVD Risk Toolkit
 *
 * Provides integration with electronic medical record systems
 * via FHIR API, allowing data import from and export to EMR systems.
 */

import XSSProtection from '../utils/xss-protection.js';
import ErrorDetectionSystem from '../utils/error-detection-system.js';

class EMRConnector {
 // EMR connection state
 static #connected = false;
 static #emrType = null;
 static #endpoint = null;
 static #authType = null;
 static #authToken = null;

 /**
 * Initialize EMR connection
 * @param {Object} options - Connection options
 * @returns {Promise<boolean>} - Whether initialization was successful
 */
 static async initialize(options) {
 try {
 console.log('Initializing EMR connection...');

 // Reset current connection
 this.#connected = false;

 // Extract options
 const {
 emrType,
 endpoint,
 authType,
 authToken
 } = options || {};

 // Validate options
 if (!emrType || !endpoint) {
 throw new Error('EMR type and endpoint are required');
 }
 } catch (error) {
 console.error(`Error initializing EMR connection: ${error.message}`);
 }
 }
}
```

```

 // Validate endpoint URL
 try {
 new URL(endpoint);
 } catch (error) {
 throw new Error('Invalid endpoint URL');
 }

 // Store connection settings
 this.#emrType = emrType;
 this.#endpoint = endpoint;
 this.#authType = authType || 'none';
 this.#authToken = authToken || null;

 // Test connection
 const connected = await this.testConnection();

 if (!connected) {
 throw new Error('Failed to connect to EMR system');
 }

 this.#connected = connected;
 return true;
} catch (error) {
 console.error('Error initializing EMR connection:', error);

 // Show error notification
 if (ErrorDetectionSystem) {
 ErrorDetectionSystem.showErrorNotification(
 new Error(`Failed to connect to EMR system: ${error.message}`),
 'EMR Connection Error'
);
 }
}

return false;
}

/**
 * Test EMR connection
 * @returns {Promise<boolean>} - Whether connection is working
 */
static async testConnection() {
 try {
 // Skip if no endpoint

```

```

 if (!this.#endpoint) {
 return false;
 }

 // Create connection test request
 const response = await this.#sendRequest('metadata', {
 method: 'GET'
 });

 // Check if response is valid
 return response && response.resourceType === 'CapabilityStatement';
 } catch (error) {
 console.error('Error testing EMR connection:', error);
 return false;
 }
}

/**
 * Check if connected to EMR
 * @returns {boolean} - Whether connected
 */
static isConnected() {
 return this.#connected;
}

/**
 * Get EMR type
 * @returns {string|null} - EMR type
 */
static getEMRType() {
 return this.#emrType;
}

/**
 * Get patient data from EMR
 * @param {string} patientId - Patient ID
 * @returns {Promise<Object>} - Patient data
 */
static async getPatientData(patientId) {
 try {
 // Check if connected
 if (!this.#connected) {
 throw new Error('Not connected to EMR system');
 }
 }

```

```
// Validate patient ID
if (!patientId) {
 throw new Error('Patient ID is required');
}

// Sanitize patient ID to prevent XSS/injection
const safePatientId = XSSProtection.sanitize(patientId);

// Fetch patient data
const patient = await this.#sendRequest(`Patient/${safePatientId}` , {
 method: 'GET'
});

if (!patient || patient.resourceType !== 'Patient') {
 throw new Error('Invalid patient data returned from EMR');
}

// Fetch patient's observations
const observations = await this.#getPatientObservations(safePatientId);

// Fetch patient's conditions
const conditions = await this.#getPatientConditions(safePatientId);

// Convert to toolkit format
const patientData = this.#mapFHIRToPatientData(patient, observations, conditions);

return {
 success: true,
 patientData,
 fhir: {
 patient,
 observations,
 conditions
 }
};
} catch (error) {
 console.error('Error getting patient data from EMR:', error);
}

return {
 success: false,
 error: error.message || 'Unknown error retrieving patient data',
 patientId
};
```

```

 }

}

/***
 * Save patient data to EMR
 * @param {Object} data - Patient data
 * @returns {Promise<Object>} - Save result
 */
static async savePatientData(data) {
 try {
 // Check if connected
 if (!this.#connected) {
 throw new Error('Not connected to EMR system');
 }

 // Convert to FHIR format
 const fhirData = this.#mapPatientDataToFHIR(data);

 // Save patient resource
 const savedPatient = await this.#sendRequest(`Patient/${fhirData.patient.id}` , {
 method: 'PUT',
 body: JSON.stringify(fhirData.patient)
 });

 // Save observations
 const savedObservations = [];

 for (const observation of fhirData.observations) {
 const savedObs = await this.#sendRequest(`Observation/${observation.id}` , {
 method: 'PUT',
 body: JSON.stringify(observation)
 });

 savedObservations.push(savedObs);
 }

 // Save conditions
 const savedConditions = [];

 for (const condition of fhirData.conditions) {
 const savedCond = await this.#sendRequest(`Condition/${condition.id}` , {
 method: 'PUT',
 body: JSON.stringify(condition)
 });
 }
 }
}

```

```

 savedConditions.push(savedCond);
 }

 // Save risk assessment
 const savedRiskAssessment = await this.#sendRequest(`RiskAssessment/${fhirData.risk
 method: 'PUT',
 body: JSON.stringify(fhirData.riskAssessment)
 });

 return {
 success: true,
 savedPatient,
 savedObservations,
 savedConditions,
 savedRiskAssessment
 };
} catch (error) {
 console.error('Error saving patient data to EMR:', error);

 return {
 success: false,
 error: error.message || 'Unknown error saving patient data'
 };
}

}

/**
 * Search for patients in EMR
 * @param {Object} criteria - Search criteria
 * @returns {Promise<Object>} - Search results
 */
static async searchPatients(criteria) {
 try {
 // Check if connected
 if (!this.#connected) {
 throw new Error('Not connected to EMR system');
 }

 // Build search parameters
 const params = new URLSearchParams();

 if (criteria.name) {
 params.append('name', criteria.name);
 }
 }
}

```

```

 }

 if (criteria.identifier) {
 params.append('identifier', criteria.identifier);
 }

 if (criteria.birthDate) {
 params.append('birthdate', criteria.birthDate);
 }

 if (criteria.gender) {
 params.append('gender', criteria.gender);
 }

 // Send search request
 const results = await this.#sendRequest(`Patient?${params.toString()}`, {
 method: 'GET'
 });

 // Check if response is valid
 if (!results || results.resourceType !== 'Bundle') {
 throw new Error('Invalid search results returned from EMR');
 }

 // Map results to simplified format
 const patients = results.entry.map(entry => {
 const resource = entry.resource;

 return {
 id: resource.id,
 name: this.#getPatientName(resource),
 gender: resource.gender,
 birthDate: resource.birthDate,
 age: this.#calculateAgeFromBirthDate(resource.birthDate)
 };
 });
}

return {
 success: true,
 patients,
 total: results.total,
 hasMore: results.link.some(link => link.relation === 'next')
};
} catch (error) {
}

```

```

 console.error('Error searching patients in EMR:', error);

 return {
 success: false,
 error: error.message || 'Unknown error searching patients',
 criteria
 };
}

/**
 * Get patient observations from EMR
 * @param {string} patientId - Patient ID
 * @returns {Promise<Array<Object>>} - Patient observations
 * @private
 */
static async #getPatientObservations(patientId) {
 try {
 // Build search parameters
 const params = new URLSearchParams();
 params.append('subject', `Patient/${patientId}`);
 params.append('category', 'vital-signs,laboratory');
 params.append('_count', '100');
 params.append('_sort', '-date');

 // Send search request
 const results = await this.#sendRequest(`Observation?${params.toString()}`, {
 method: 'GET'
 });

 // Check if response is valid
 if (!results || results.resourceType !== 'Bundle') {
 throw new Error('Invalid observation results returned from EMR');
 }

 // Extract observations
 return results.entry.map(entry => entry.resource);
 } catch (error) {
 console.error('Error getting patient observations from EMR:', error);
 return [];
 }
}

```

```

* Get patient conditions from EMR
* @param {string} patientId - Patient ID
* @returns {Promise<Array<Object>>} - Patient conditions
* @private
*/
static async #getPatientConditions(patientId) {
 try {
 // Build search parameters
 const params = new URLSearchParams();
 params.append('subject', `Patient/${patientId}`);
 params.append('clinical-status', 'active');
 params.append('_count', '100');

 // Send search request
 const results = await this.#sendRequest(`Condition?${params.toString()}`, {
 method: 'GET'
 });

 // Check if response is valid
 if (!results || results.resourceType !== 'Bundle') {
 throw new Error('Invalid condition results returned from EMR');
 }

 // Extract conditions
 return results.entry.map(entry => entry.resource);
 } catch (error) {
 console.error('Error getting patient conditions from EMR:', error);
 return [];
 }
}

/**
 * Map FHIR resources to patient data
 * @param {Object} patient - FHIR Patient resource
 * @param {Array<Object>} observations - FHIR Observation resources
 * @param {Array<Object>} conditions - FHIR Condition resources
 * @returns {Object} - Patient data
 * @private
*/
static #mapFHIRTToPatientData(patient, observations, conditions) {
 const data = {};

 // Extract basic patient information
 data.patientId = patient.id;

```

```

if (patient.gender) {
 data.sex = patient.gender;
}

// Calculate age from birth date
if (patient.birthDate) {
 data.age = this.#calculateAgeFromBirthDate(patient.birthDate);
}

// Extract smoking status
if (patient.extension) {
 const smokingExtension = patient.extension.find(
 ext => ext.url === 'http://hl7.org/fhir/StructureDefinition/patient-smokingStatus');
}

if (smokingExtension && smokingExtension.valueCodeableConcept) {
 const smokingCode = smokingExtension.valueCodeableConcept.coding[0]?.code;
 data.isSmoker = ['current', 'active'].includes(smokingCode.toLowerCase());

 // Map to QRISK3 smoking status
 if (smokingCode.toLowerCase() === 'current') {
 data.smokingStatus = 'light';
 } else if (smokingCode.toLowerCase() === 'former') {
 data.smokingStatus = 'ex';
 } else {
 data.smokingStatus = 'non';
 }
}
}

// Process observations
for (const obs of observations) {
 // Skip if no code or value
 if (!obs.code || !obs.code.coding || !obs.valueQuantity) {
 continue;
 }

 // Get code and value
 const code = obs.code.coding[0]?.code;
 const value = obs.valueQuantity.value;
 const unit = obs.valueQuantity.unit;

 // Skip if no code or value
}

```

```
if (!code || !value) {
 continue;
}

// Map code to data field
switch (code) {
 case '8480-6': // Systolic BP
 data.systolicBP = parseFloat(value);
 break;

 case '8462-4': // Diastolic BP
 data.diastolicBP = parseFloat(value);
 break;

 case '14647-2': // Total cholesterol
 data.totalCholesterol = parseFloat(value);

 // Convert if needed
 if (unit === 'mg/dL') {
 data.totalCholesterol = data.totalCholesterol / 38.67;
 }
 break;

 case '14646-4': // HDL
 data.hdl = parseFloat(value);

 // Convert if needed
 if (unit === 'mg/dL') {
 data.hdl = data.hdl / 38.67;
 }
 break;

 case '13457-7': // LDL
 data.ldl = parseFloat(value);

 // Convert if needed
 if (unit === 'mg/dL') {
 data.ldl = data.ldl / 38.67;
 }
 break;

 case '14927-8': // Triglycerides
 data.triglycerides = parseFloat(value);
}
```

```

 // Convert if needed
 if (unit === 'mg/dL') {
 data.triglycerides = data.triglycerides / 88.5;
 }
 break;
 }
}

// Process conditions
for (const cond of conditions) {
 // Skip if no code
 if (!cond.code || !cond.code.coding) {
 continue;
 }

 // Get code
 const code = cond.code.coding[0]?.code;

 // Skip if no code
 if (!code) {
 continue;
 }

 // Map code to data field
 switch (code) {
 case '44054006': // Diabetes
 data.hasDiabetes = true;

 // Try to determine type
 const diabetesType = cond.code.coding.find(c =>
 c.code === '46635009' || c.code === '44054006'
);

 if (diabetesType) {
 data.diabetesStatus = diabetesType.code === '46635009' ? 'type1' : 'typ
 } else {
 data.diabetesStatus = 'type2'; // Default to type 2
 }
 resourceType: 'Observation',
 id: `obs-ldl-${new Date().getTime()}`,
 status: 'final',
 category: [
 {
 // Try to load patient data into medication form
 const medicationForm = document.getElementById('medication-form');

```

```

 if (medicationForm) {
 this.#populateForm(medicationForm, data.patientData || data);
 }

 // Try to Load FRS data into FRS form
 const frsForm = document.getElementById('frs-form');
 if (frsForm && data.frs) {
 this.#populateForm(frsForm, data.frs);
 }

 // Try to Load QRISK data into QRISK form
 const qriskForm = document.getElementById('qrisk-form');
 if (qriskForm && data.qrisk) {
 this.#populateForm(qriskForm, data.qrisk);
 }

 // Save results to session storage if available
 if (data.frsResult) {
 sessionStorage.setItem('frs_result', JSON.stringify(data.frsResult));
 }

 if (data.qriskResult) {
 sessionStorage.setItem('qrisk_result', JSON.stringify(data.qriskResult));
 }

 if (data.recommendations) {
 sessionStorage.setItem('recommendations', JSON.stringify(data.recommendations))
 }

 return true;
 } catch (error) {
 console.error('Error loading imported data:', error);
 return false;
 }
}

/**
 * Read file content
 * @param {File} file - File to read
 * @returns {Promise<string>} - File content
 * @private
 */
static async #readFile(file) {
 return new Promise((resolve, reject) => {

```

```

const reader = new FileReader();

reader.onload = (event) => {
 resolve(event.target.result);
};

reader.onerror = (error) => {
 reject(new Error(`Error reading file: ${error.message}`));
};

reader.readAsText(file);
});

}

/***
 * Process JSON data
 * @param {string} content - JSON content
 * @param {boolean} decrypt - Whether to decrypt data
 * @returns {Promise<Object>} - Processed data
 * @private
 */
static async #processJSON(content, decrypt = false) {
 try {
 let data = JSON.parse(content);

 // Decrypt if requested
 if (decrypt && data.encrypted) {
 if (!await SecureStorage.initialize()) {
 throw new Error('Failed to initialize secure storage for decryption');
 }
 }

 // Extract encrypted data and decrypt
 const decrypted = await SecureStorage.decrypt(data.data);
 data = JSON.parse(decrypted);
 }

 return data;
} catch (error) {
 throw new Error(`Error processing JSON: ${error.message}`);
}
}

/***
 * Process CSV data
*/

```

```

* @param {string} content - CSV content
* @returns {Promise<Object>} - Processed data
* @private
*/
static async #processCSV(content) {
 try {
 // Try to import Papa Parse
 const Papa = (await import('papaparse')).default;

 // Parse CSV
 const parsedData = Papa.parse(content, {
 header: true,
 dynamicTyping: true,
 skipEmptyLines: true
 });

 if (parsedData.errors && parsedData.errors.length > 0) {
 const errorMessage = parsedData.errors.map(e => e.message).join('; ');
 throw new Error(`CSV parsing errors: ${errorMessage}`);
 }

 // Convert array data to object (assuming one patient per file)
 if (parsedData.data && parsedData.data.length > 0) {
 return parsedData.data[0];
 } else {
 throw new Error('No data found in CSV file');
 }
 } catch (error) {
 throw new Error(`Error processing CSV: ${error.message}`);
 }
}

/**
 * Process Excel data
* @param {File} file - Excel file
* @returns {Promise<Object>} - Processed data
* @private
*/
static async #processExcel(file) {
 try {
 // Try to import SheetJS
 const XLSX = (await import('xlsx')).default;

 // Read Excel file

```

```

 const arrayBuffer = await file.arrayBuffer();
 const workbook = XLSX.read(arrayBuffer, { type: 'array' });

 // Get first sheet
 const sheetName = workbook.SheetNames[0];
 const sheet = workbook.Sheets[sheetName];

 // Convert to JSON
 const data = XLSX.utils.sheet_to_json(sheet, { raw: false });

 // Return first row (assuming one patient per file)
 if (data && data.length > 0) {
 return data[0];
 } else {
 throw new Error('No data found in Excel file');
 }
} catch (error) {
 throw new Error(`Error processing Excel: ${error.message}`);
}

/**
 * Process FHIR data
 * @param {string} content - FHIR content
 * @returns {Promise<Object>} - Processed data
 * @private
 */
static async #processFHIR(content) {
 try {
 const fhirData = JSON.parse(content);

 // Check if valid FHIR resource
 if (!fhirData.resourceType) {
 throw new Error('Invalid FHIR resource (missing resourceType)');
 }

 // Handle different resource types
 if (fhirData.resourceType === 'Patient') {
 return this.#mapFHIRPatientToData(fhirData);
 } else if (fhirData.resourceType === 'Bundle') {
 return this.#mapFHIRBundleToData(fhirData);
 } else {
 throw new Error(`Unsupported FHIR resource type: ${fhirData.resourceType}`);
 }
 }
}

```

```

 } catch (error) {
 throw new Error(`Error processing FHIR: ${error.message}`);
 }
}

/**
 * Process HL7 data
 * @param {string} content - HL7 content
 * @returns {Promise<Object>} - Processed data
 * @private
 */
static async #processHL7(content) {
 try {
 // Simple HL7 parser (very basic)
 const segments = content.split('\n');
 const data = {};

 for (const segment of segments) {
 const fields = segment.split('|');
 const segmentType = fields[0];

 if (segmentType === 'PID') {
 // Process patient identification
 data.patientId = fields[3] || '';

 // Extract name from field 5 (Last^First^Middle)
 if (fields[5]) {
 const nameParts = fields[5].split('^');
 data.lastName = nameParts[0] || '';
 data.firstName = nameParts[1] || '';
 }
 }

 // Extract sex from field 8
 if (fields[8]) {
 data.sex = fields[8].toLowerCase() === 'm' ? 'male' :
 fields[8].toLowerCase() === 'f' ? 'female' : '';
 }

 // Extract DOB from field 7
 if (fields[7]) {
 try {
 const dob = new Date(fields[7]);
 const ageDate = new Date(Date.now() - dob.getTime());
 data.age = Math.abs(ageDate.getUTCFullYear() - 1970);
 }
 }
 }
 }
}

```

```
 } catch (e) {
 console.error('Error calculating age from DOB:', e);
 }
 }

} else if (segmentType === 'OBX') {
 // Process observation/result
 const observationId = fields[3] || '';
 const observationValue = fields[5] || '';

 // Map observation IDs to data fields
 switch (observationId) {
 case 'SBP':
 case '8480-6':
 data.systolicBP = parseFloat(observationValue);
 break;

 case 'DBP':
 case '8462-4':
 data.diastolicBP = parseFloat(observationValue);
 break;

 case 'TCHOL':
 case '14647-2':
 data.totalCholesterol = parseFloat(observationValue);
 break;

 case 'HDL':
 case '14646-4':
 data.hdl = parseFloat(observationValue);
 break;

 case 'LDL':
 case '13457-7':
 data.ldl = parseFloat(observationValue);
 break;

 case 'TRIG':
 case '14927-8':
 data.triglycerides = parseFloat(observationValue);
 break;
 }
}
```

```

 return data;
 } catch (error) {
 throw new Error(`Error processing HL7: ${error.message}`);
 }
}

/**
 * Map FHIR Patient resource to data
 * @param {Object} patient - FHIR Patient resource
 * @returns {Object} - Mapped data
 * @private
 */
static #mapFHIRPatientToData(patient) {
 const data = {};

 // Extract basic patient information
 if (patient.gender) {
 data.sex = patient.gender;
 }

 // Calculate age from birth date
 if (patient.birthDate) {
 try {
 const dob = new Date(patient.birthDate);
 const ageDate = new Date(Date.now() - dob.getTime());
 data.age = Math.abs(ageDate.getUTCFullYear() - 1970);
 } catch (e) {
 console.error('Error calculating age from birthDate:', e);
 }
 }

 // Extract smoking status
 if (patient.extension) {
 const smokingExtension = patient.extension.find(
 ext => ext.url === 'http://hl7.org/fhir/StructureDefinition/patient-smokingStatus'
);

 if (smokingExtension && smokingExtension.valueCodeableConcept) {
 const smokingCode = smokingExtension.valueCodeableConcept.coding[0]?.code;
 data.isSmoker = ['current', 'active'].includes(smokingCode.toLowerCase());
 }
 }
}

return data;

```

```

}

/**
 * Map FHIR Bundle to data
 * @param {Object} bundle - FHIR Bundle
 * @returns {Object} - Mapped data
 * @private
 */
static #mapFHIRBundleToData(bundle) {
 if (!bundle.entry || !Array.isArray(bundle.entry)) {
 throw new Error('Invalid FHIR Bundle (missing or invalid entry array)');
 }

 const data = {};

 // Process patient resource
 const patientEntry = bundle.entry.find(
 entry => entry.resource && entry.resource.resourceType === 'Patient'
);

 if (patientEntry) {
 Object.assign(data, this.#mapFIRPatientToData(patientEntry.resource));
 }

 // Process observation resources
 const observations = bundle.entry.filter(
 entry => entry.resource && entry.resource.resourceType === 'Observation'
);

 for (const obsEntry of observations) {
 const obs = obsEntry.resource;

 // Skip if no code or value
 if (!obs.code || !obs.code.coding || !obs.valueQuantity) {
 continue;
 }

 // Get code and value
 const code = obs.code.coding[0]?.code;
 const value = obs.valueQuantity.value;

 // Skip if no code or value
 if (!code || !value) {
 continue;
 }
 }
}

```

```

}

// Map code to data field
switch (code) {
 case '8480-6': // Systolic BP
 data.systolicBP = parseFloat(value);
 break;

 case '8462-4': // Diastolic BP
 data.diastolicBP = parseFloat(value);
 break;

 case '14647-2': // Total cholesterol
 data.totalCholesterol = parseFloat(value);
 break;

 case '14646-4': // HDL
 data.hdl = parseFloat(value);
 break;

 case '13457-7': // LDL
 data.ldl = parseFloat(value);
 break;

 case '14927-8': // Triglycerides
 data.triglycerides = parseFloat(value);
 break;
}

// Process condition resources
const conditions = bundle.entry.filter(
 entry => entry.resource && entry.resource.resourceType === 'Condition'
);

for (const condEntry of conditions) {
 const cond = condEntry.resource;

 // Skip if no code
 if (!cond.code || !cond.code.coding) {
 continue;
 }

 // Get code

```

```

const code = cond.code.coding[0]?.code;

// Skip if no code
if (!code) {
 continue;
}

// Map code to data field
switch (code) {
 case '44054006': // Diabetes
 data.hasDiabetes = true;
 break;

 case '22298006': // Atrial fibrillation
 data.atrialFibrillation = true;
 break;

 case '73211009': // Rheumatoid arthritis
 data.rheumatoidArthritis = true;
 break;

 case '396332003': // Chronic kidney disease
 data.chronicKidneyDisease = true;
 break;
}

return data;
}

/**
 * Validate imported data
 * @param {Object} data - Data to validate
 * @throws {Error} - If validation fails
 * @private
 */
static #validateImportedData(data) {
 if (!data || typeof data !== 'object') {
 throw new Error('Invalid data format (not an object)');
 }

 // Check for required fields for basic functionality
 const requiredFields = ['age', 'sex'];

```

```

 for (const field of requiredFields) {
 if (data[field] === undefined) {
 throw new Error(`Missing required field: ${field}`);
 }
 }

 }

 /**
 * Prepare data for export
 * @param {Object} data - Data to export
 * @param {boolean} includeRecommendations - Whether to include recommendations
 * @returns {Object} - Prepared data
 * @private
 */
 static #prepareExportData(data, includeRecommendations = true) {
 const exportData = {
 exportDate: new Date().toISOString(),
 patientData: { ...data }
 };

 // Add saved results if available
 try {
 const frsResult = sessionStorage.getItem('frs_result');
 if (frsResult) {
 exportData.frsResult = JSON.parse(frsResult);
 }

 const qriskResult = sessionStorage.getItem('qrisk_result');
 if (qriskResult) {
 exportData.qriskResult = JSON.parse(qriskResult);
 }

 // Add recommendations if requested
 if (includeRecommendations) {
 const recommendations = sessionStorage.getItem('recommendations');
 if (recommendations) {
 exportData.recommendations = JSON.parse(recommendations);
 }
 }
 } catch (error) {
 console.error('Error retrieving saved results:', error);
 }

 return exportData;
 }
}

```

```
}

/**
 * Export data as JSON
 * @param {Object} data - Data to export
 * @param {boolean} encrypt - Whether to encrypt data
 * @returns {Promise<Object>} - Export result
 * @private
 */
static async #exportJSON(data, encrypt = false) {
 try {
 let exportData = data;

 // Encrypt if requested
 if (encrypt) {
 if (!await SecureStorage.initialize()) {
 throw new Error('Failed to initialize secure storage for encryption');
 }
 }

 // Encrypt data
 const encrypted = await SecureStorage.encrypt(JSON.stringify(data));

 // Create wrapper object
 exportData = {
 encrypted: true,
 data: encrypted,
 encryptionVersion: 1,
 exportDate: new Date().toISOString()
 };
 }

 // Convert to JSON string
 const jsonString = JSON.stringify(exportData, null, 2);

 // Create blob and URL
 const blob = new Blob([jsonString], { type: 'application/json' });
 const url = URL.createObjectURL(blob);

 // Create filename
 const timestamp = new Date().toISOString().replace(/[-.:]/g, '').substring(0, 15);
 const filename = `cvd-risk-data-${timestamp}.json`;

 // Download file
 this.#downloadFile(url, filename);
}
```

```

 return {
 url,
 filename,
 format: 'json',
 encrypted
 };
 } catch (error) {
 throw new Error(`Error exporting JSON: ${error.message}`);
 }
}

/**
 * Export data as CSV
 * @param {Object} data - Data to export
 * @returns {Promise<Object>} - Export result
 * @private
 */
static async #exportCSV(data) {
 try {
 // Try to import Papa Parse
 const Papa = (await import('papaparse')).default;

 // Flatten data for CSV export
 const flatData = this.#flattenObjectForCSV(data);

 // Convert to CSV
 const csvString = Papa.unparse([flatData], {
 header: true,
 quotes: true,
 quoteChar: "'",
 escapeChar: "'",
 delimiter: ','
 });

 // Create blob and URL
 const blob = new Blob([csvString], { type: 'text/csv;charset=utf-8' });
 const url = URL.createObjectURL(blob);

 // Create filename
 const timestamp = new Date().toISOString().replace(/[-.:]/g, '').substring(0, 15);
 const filename = `cvd-risk-data-${timestamp}.csv`;

 // Download file
 }
}

```

```
 this.#downloadFile(url, filename);

 return {
 url,
 filename,
 format: 'csv'
 };
} catch (error) {
 throw new Error(`Error exporting CSV: ${error.message}`);
}
}

/**
 * Export data as Excel
 * @param {Object} data - Data to export
 * @returns {Promise<Object>} - Export result
 * @private
 */
static async #exportExcel(data) {
 try {
 // Try to import SheetJS
 const XLSX = (await import('xlsx')).default;

 // Flatten data for Excel export
 const flatData = this.#flattenObjectForCSV(data);

 // Create workbook
 const workbook = XLSX.utils.book_new();

 // Create worksheet
 const worksheet = XLSX.utils.json_to_sheet([flatData]);

 // Add worksheet to workbook
 XLSX.utils.book_append_sheet(workbook, worksheet, 'Patient Data');

 // Generate Excel file
 const excelData = XLSX.write(workbook, { bookType: 'xlsx', type: 'array' });

 // Create blob and URL
 const blob = new Blob([excelData], { type: 'application/vnd.openxmlformats-officedc
 const url = URL.createObjectURL(blob);

 // Create filename
 const timestamp = new Date().toISOString().replace(/[-:.]/g, '').substring(0, 15);
```

```

const filename = `cvd-risk-data-${timestamp}.xlsx`;

// Download file
this.#downloadFile(url, filename);

return {
 url,
 filename,
 format: 'excel'
};
} catch (error) {
 throw new Error(`Error exporting Excel: ${error.message}`);
}

}

/**
 * Export data as PDF
 * @param {Object} data - Data to export
 * @returns {Promise<Object>} - Export result
 * @private
 */
static async #exportPDF(data) {
 try {
 // Try to import PDF generator
 const PDFGenerator = (await import('./pdf-generator.js')).default;

 // Generate PDF
 const result = await PDFGenerator.generateReport(data);

 return {
 ...result,
 format: 'pdf'
 };
 } catch (error) {
 throw new Error(`Error exporting PDF: ${error.message}`);
 }
}

/**
 * Export data as FHIR
 * @param {Object} data - Data to export
 * @returns {Promise<Object>} - Export result
 * @private
 */

```

```

static async #exportFHIR(data) {
 try {
 // Create FHIR Bundle
 const bundle = {
 resourceType: 'Bundle',
 type: 'collection',
 entry: []
 };

 // Add patient resource
 bundle.entry.push({
 resource: this.#createFHIRPatientResource(data.patientData || data)
 });

 // Add observation resources
 const observations = this.#createFHIRObservationResources(data.patientData || data)
 observations.forEach(obs => {
 bundle.entry.push({ resource: obs });
 });

 // Add condition resources
 const conditions = this.#createFHIRConditionResources(data.patientData || data);
 conditions.forEach(cond => {
 bundle.entry.push({ resource: cond });
 });

 // Add risk assessment resources
 if (data.frsResult) {
 bundle.entry.push({
 resource: this.#createFIRRiskAssessmentResource(data.frsResult, 'framingham')
 });
 }

 if (data.qriskResult) {
 bundle.entry.push({
 resource: this.#createFIRRiskAssessmentResource(data.qriskResult, 'qrisk3')
 });
 }

 // Convert to JSON string
 const jsonString = JSON.stringify(bundle, null, 2);

 // Create blob and URL
 const blob = new Blob([jsonString], { type: 'application/fhir+json' });
 }
}

```

```

 const url = URL.createObjectURL(blob);

 // Create filename
 const timestamp = new Date().toISOString().replace(/[-:.]/g, '').substring(0, 15);
 const filename = `cvd-risk-fhir-${timestamp}.json`;

 // Download file
 this.#downloadFile(url, filename);

 return {
 url,
 filename,
 format: 'fhir'
 };
 } catch (error) {
 throw new Error(`Error exporting FHIR: ${error.message}`);
 }
}

/**
 * Create FHIR Patient resource
 * @param {Object} data - Patient data
 * @returns {Object} - FHIR Patient resource
 * @private
 */
static #createFHIRPatientResource(data) {
 const patient = {
 resourceType: 'Patient',
 id: `patient-${new Date().getTime()}`,
 meta: {
 profile: ['http://hl7.org/fhir/StructureDefinition/Patient']
 },
 active: true,
 gender: data.sex || 'unknown',
 extension: []
 };

 // Add smoking status extension
 if (data.isSmoker !== undefined) {
 patient.extension.push({
 url: 'http://hl7.org/fhir/StructureDefinition/patient-smokingStatus',
 valueCodeableConcept: {
 coding: [
 {

```

```

 system: 'http://snomed.info/sct',
 code: data.isSmoker ? '449868002' : '266919005',
 display: data.isSmoker ? 'Current smoker' : 'Never smoked'
 }
],
text: data.isSmoker ? 'Current smoker' : 'Never smoked'
})
});

}

// Calculate birth date from age (approximate)
if (data.age) {
 const birthYear = new Date().getFullYear() - data.age;
 patient.birthDate = `${birthYear}-01-01`;
}

return patient;
}

/**
 * Create FHIR Observation resources
 * @param {Object} data - Patient data
 * @returns {Array<Object>} - FHIR Observation resources
 * @private
 */
static #createFHIRObservationResources(data) {
 const observations = [];
 const patientId = `patient-${new Date().getTime()}`;
 const obsDate = new Date().toISOString();

 // Systolic BP
 if (data.systolicBP) {
 observations.push({
 resourceType: 'Observation',
 id: `obs-sbp-${new Date().getTime()}`,
 status: 'final',
 category: [
 {
 coding: [
 {
 system: 'http://terminology.hl7.org/CodeSystem/observation-category',
 code: 'vital-signs',
 display: 'Vital Signs'
 }
]
 }
]
 });
 }
}

```

```

]
 }
],
code: {
 coding: [
 {
 system: 'http://loinc.org',
 code: '8480-6',
 display: 'Systolic blood pressure'
 }
],
 text: 'Systolic blood pressure'
},
subject: {
 reference: `Patient/${patientId}`
},
effectiveDateTime: obsDate,
valueQuantity: {
 value: data.systolicBP,
 unit: 'mmHg',
 system: 'http://unitsofmeasure.org',
 code: 'mm[Hg]'
}
});
}

// Diastolic BP
if (data.diastolicBP) {
 observations.push({
 resourceType: 'Observation',
 id: `obs-dbp-${new Date().getTime()}`,
 status: 'final',
 category: [
 {
 coding: [
 {
 system: 'http://terminology.hl7.org/CodeSystem/observation-category',
 code: 'vital-signs',
 display: 'Vital Signs'
 }
]
 }
],
 code: {

```

```

 coding: [
 {
 system: 'http://loinc.org',
 code: '8462-4',
 display: 'Diastolic blood pressure'
 }
],
 text: 'Diastolic blood pressure'
 },
 subject: {
 reference: `Patient/${patientId}`
 },
 effectiveDateTime: obsDate,
 valueQuantity: {
 value: data.diastolicBP,
 unit: 'mmHg',
 system: 'http://unitsofmeasure.org',
 code: 'mm[Hg]'
 }
}
});
```

}

// Total cholesterol

```

if (data.totalCholesterol) {
 observations.push({
 resourceType: 'Observation',
 id: `obs-tc-${new Date().getTime()}`,
 status: 'final',
 category: [
 {
 coding: [
 {
 system: 'http://terminology.hl7.org/CodeSystem/observation-category',
 code: 'laboratory',
 display: 'Laboratory'
 }
]
 }
],
 code: {
 coding: [
 {
 system: 'http://loinc.org',
 code: '14647-2',

```

```

 display: 'Total cholesterol'
 }
],
text: 'Total cholesterol'
},
subject: {
 reference: `Patient/${patientId}`
},
effectiveDateTime: obsDate,
valueQuantity: {
 value: data.totalCholesterol,
 unit: 'mmol/L',
 system: 'http://unitsofmeasure.org',
 code: 'mmol/L'
}
});
}

// HDL
if (data.hdl) {
 observations.push({
 resourceType: 'Observation',
 id: `obs-hdl-${new Date().getTime()}`,
 status: 'final',
 category: [
 {
 coding: [
 {
 system: 'http://terminology.hl7.org/CodeSystem/observation-category',
 code: 'laboratory',
 display: 'Laboratory'
 }
]
 }
],
 code: {
 coding: [
 {
 system: 'http://loinc.org',
 code: '14646-4',
 display: 'HDL cholesterol'
 }
],
 text: 'HDL cholesterol'
 }
 })
}

```

```

 },
 subject: {
 reference: `Patient/${patientId}`
 },
 effectiveDateTime: obsDate,
 valueQuantity: {
 value: data.hdl,
 unit: 'mmol/L',
 system: 'http://unitsofmeasure.org',
 code: 'mmol/L'
 }
 });
 }
 }

// LDL
if (data.ldl) {
 observations.push({
 resourceType: 'Observation',
 id: `obs-lld-${new Date().getTime()}`,
 status: 'final',
 category: [
 {
 coding: [
 {
 system: 'http://## 4. Data Management

```

### ### 4.1 data-import-export.js

```

```javascript
/**
 * Data Import/Export for CVD Risk Toolkit
 *
 * Handles import and export of patient data and risk assessment
 * results in various formats, including JSON, CSV, Excel, FHIR,
 * and HL7.
 */

import SecureStorage from '../utils/secure-storage.js';
import XSSProtection from '../utils/xss-protection.js';

class DataImportExport {
  /**
   * Import data from file
   * @param {File} file - File to import

```

```
* @param {string} format - File format
* @param {boolean} decrypt - Whether to decrypt imported data
* @returns {Promise<Object>} - Import result
*/
static async importData(file, format, decrypt = false) {
    try {
        console.log(`Importing data from ${file.name} (${format})...`);

        // Read file
        const fileContent = await this.#readFile(file);

        // Process based on format
        let data;

        switch (format.toLowerCase()) {
            case 'json':
                data = await this.#processJSON(fileContent, decrypt);
                break;

            case 'csv':
                data = await this.#processCSV(fileContent);
                break;

            case 'excel':
                data = await this.#processExcel(file);
                break;

            case 'fhir':
                data = await this.#processFHIR(fileContent);
                break;

            case 'hl7':
                data = await this.#processHL7(fileContent);
                break;

            default:
                throw new Error(`Unsupported format: ${format}`);
        }

        // Validate imported data
        this.#validateImportedData(data);

        // Sanitize data to prevent XSS
        const sanitizedData = XSSProtection.sanitizeObject(data);
    }
}
```

```

        return {
            success: true,
            data: sanitizedData,
            format,
            originalFile: file.name
        };
    } catch (error) {
        console.error('Error importing data:', error);

        return {
            success: false,
            error: error.message || 'Unknown error during import',
            format,
            originalFile: file.name
        };
    }
}

/**
 * Export data
 * @param {Object} data - Data to export
 * @param {string} format - Export format
 * @param {boolean} encrypt - Whether to encrypt exported data
 * @param {boolean} includeRecommendations - Whether to include recommendations
 * @returns {Promise<Object>} - Export result
 */
static async exportData(data, format, encrypt = false, includeRecommendations = true) {
    try {
        console.log(`Exporting data as ${format}...`);

        // Prepare data for export
        const exportData = this.#prepareExportData(data, includeRecommendations);

        // Process based on format
        let result;

        switch (format.toLowerCase()) {
            case 'json':
                result = await this.#exportJSON(exportData, encrypt);
                break;

            case 'csv':
                result = await this.#exportCSV(exportData);
        }
    } catch (error) {
        console.error('Error exporting data:', error);
    }
}

```

```

        break;

    case 'excel':
        result = await this.#exportExcel(exportData);
        break;

    case 'pdf':
        result = await this.#exportPDF(exportData);
        break;

    case 'fhir':
        result = await this.#exportFHIR(exportData);
        break;

    default:
        throw new Error(`Unsupported format: ${format}`);
    }

    return {
        success: true,
        result,
        format
    };
} catch (error) {
    console.error('Error exporting data:', error);

    return {
        success: false,
        error: error.message || 'Unknown error during export',
        format
    };
}

/**
 * Load imported data into forms
 * @param {Object} data - Imported data
 * @returns {Promise<boolean>} - Whether Loading was successful
 */
static async loadImportedData(data) {
    try {
        console.log('Loading imported data into forms...');

        if (!data) {

```

```
        throw new Error('No data to load');
    }

    // Try to Load patient data into medication form
    const medicationForm = document.getElementById('medication-form'); // Tr
    const frsForm = document.getElementById('frs-form');
    if (frsForm) {
        const formData = new FormData(frsForm);

        for (const [key, value] of formData.entries()) {
            patientData[key] = value;
        }
    }

    // Convert checkbox values
    const checkboxes = frsForm.querySelectorAll('input[type="checkbox"]');
    checkboxes.forEach(checkbox => {
        patientData[checkbox.name] = checkbox.checked;
    });

    return patientData;
}

// Try to extract from QRISK form
const qriskForm = document.getElementById('qrisk-form');
if (qriskForm) {
    const formData = new FormData(qriskForm);

    for (const [key, value] of formData.entries()) {
        patientData[key] = value;
    }
}

// Convert checkbox values
const checkboxes = qriskForm.querySelectorAll('input[type="checkbox"]');
checkboxes.forEach(checkbox => {
    patientData[checkbox.name] = checkbox.checked;
});

return patientData;
}

// Try to extract from medication form
const medicationForm = document.getElementById('medication-form');
if (medicationForm) {
    const formData = new FormData(medicationForm);
```

```

        for (const [key, value] of formData.entries()) {
            patientData[key] = value;
        }

        // Convert checkbox values
        const checkboxes = medicationForm.querySelectorAll('input[type="checkbox"]');
        checkboxes.forEach(checkbox => {
            patientData[checkbox.name] = checkbox.checked;
        });

        return patientData;
    }
} catch (error) {
    console.error('Error extracting patient data from forms:', error);
}

// If no data found, return error
return null;
}

/**
 * Create canvas element for chart
 * @param {HTMLElement} container - Chart container
 * @returns {HTMLCanvasElement} - Canvas element
 * @private
 */
static #createCanvasElement(container) {
    // Clear container
    container.innerHTML = '';

    // Create canvas
    const canvas = document.createElement('canvas');
    canvas.className = 'chart-canvas';
    container.appendChild(canvas);

    return canvas;
}

/**
 * Get guideline recommendation
 * @param {string} guideline - Guideline name
 * @param {string} riskCategory - Risk category
 * @returns {string} - Recommendation

```

```

* @private
*/
static #getGuidelineRecommendation(guideline, riskCategory) {
    if (guideline === 'ccs2021') {
        switch (riskCategory) {
            case 'low':
                return 'Consider treatment if LDL-C ≥ 5.0 mmol/L';
            case 'intermediate':
                return 'Consider treatment if LDL-C ≥ 3.5 mmol/L';
            case 'high':
                return 'Treatment recommended if LDL-C ≥ 2.0 mmol/L';
            case 'veryHigh':
                return 'Treatment recommended (target LDL-C < 1.4 mmol/L)';
            default:
                return 'Unknown risk category';
        }
    }
}

return 'Guideline not available';
}

/***
* Get risk category label
* @param {number} risk - Risk percentage
* @returns {string} - Risk category Label
* @private
*/
static #getRiskCategoryLabel(risk) {
    if (risk < 10) {
        return 'low risk';
    } else if (risk < 20) {
        return 'intermediate risk';
    } else {
        return 'high risk';
    }
}

/***
* Get risk category
* @param {number} risk - Risk percentage
* @returns {string} - Risk category
* @private
*/
static #getRiskCategory(risk) {

```

```

    if (risk < 10) {
        return 'low';
    } else if (risk < 20) {
        return 'intermediate';
    } else {
        return 'high';
    }
}

/**
 * Get intervention label
 * @param {string} intervention - Intervention type
 * @returns {string} - Label
 * @private
 */
static #getInterventionLabel(intervention) {
    switch (intervention) {
        case 'statin':
            return 'Statin Therapy';
        case 'bp-reduction':
            return 'BP Treatment';
        case 'smoking-cessation':
            return 'Smoking Cessation';
        case 'combined':
            return 'Combined Interventions';
        default:
            return intervention;
    }
}

export default ChartRenderer;

```

3.2 chart-exporter.js

javascript

```
/**  
 * Chart Exporter for CVD Risk Toolkit  
 *  
 * Provides functionality to export charts as images, PDF documents,  
 * or to copy chart data to the clipboard.  
 */  
  
class ChartExporter {  
    /**  
     * Export a chart  
     * @param {HTMLInputElement} container - Chart container  
     * @param {string} format - Export format (png, svg, pdf, excel)  
     * @returns {Promise<boolean>} - Whether export was successful  
     */  
    static async exportChart(container, format = 'png') {  
        try {  
            console.log(`Exporting chart as ${format}`);  
  
            if (!container) {  
                throw new Error('Chart container is required');  
            }  
  
            // Find canvas or container  
            const canvas = container.querySelector('canvas');  
  
            if (!canvas && format !== 'excel') {  
                throw new Error('No chart canvas found in container');  
            }  
  
            // Export based on format  
            switch (format.toLowerCase()) {  
                case 'png':  
                    return this.#exportAsPNG(canvas);  
  
                case 'svg':  
                    return this.#exportAsSVG(canvas);  
  
                case 'pdf':  
                    return this.#exportAsPDF(canvas, container);  
  
                case 'excel':  
                    return this.#exportAsExcel(container);  
            }  
        } catch (error) {  
            console.error(error);  
            return Promise.reject(error);  
        }  
    }  
}  
// Exports a chart to a file  
const fileSaver = require('file-saver');  
  
ChartExporter.prototype.#exportAsPNG = (canvas) => {  
    const data = canvas.toDataURL('image/png');  
    return fileSaver.saveAs(data, 'chart.png');  
};  
  
ChartExporter.prototype.#exportAsSVG = (canvas) => {  
    const data = canvas.outerHTML;  
    return fileSaver.saveAs(data, 'chart.svg');  
};  
  
ChartExporter.prototype.#exportAsPDF = (canvas, container) => {  
    const pdf = jsPDF();  
    pdf.addImage(canvas, 'JPEG', 0, 0, 100, 100);  
    pdf.save('chart.pdf');  
};  
  
ChartExporter.prototype.#exportAsExcel = (container) => {  
    const excel = new ExcelJS();  
    const sheet = excel.addSheet('Chart Data');  
    const data = [...container.querySelectorAll('table tr')].map((tr) => {  
        const cells = [...tr.querySelectorAll('td')];  
        return cells.map((td) => td.textContent);  
    });  
    sheet.addRows(data);  
    const blob = excel.xlsx.writeAsBlob(excel);  
    return fileSaver.saveAs(blob, 'chart.xlsx');  
};
```

```

        default:
            throw new Error(`Unsupported export format: ${format}`);
        }
    } catch (error) {
        console.error('Error exporting chart:', error);
        return false;
    }
}

/***
 * Export chart as PNG
 * @param {HTMLCanvasElement} canvas - Chart canvas
 * @returns {Promise<boolean>} - Whether export was successful
 * @private
 */
static async #exportAsPNG(canvas) {
    try {
        // Create a temporary Link
        const link = document.createElement('a');

        // Set link properties
        link.download = `chart-${new Date().toISOString().substring(0, 10)}.png`;
        link.href = canvas.toDataURL('image/png');

        // Click link to trigger download
        document.body.appendChild(link);
        link.click();
        document.body.removeChild(link);

        return true;
    } catch (error) {
        console.error('Error exporting as PNG:', error);
        return false;
    }
}

/***
 * Export chart as SVG
 * @param {HTMLCanvasElement} canvas - Chart canvas
 * @returns {Promise<boolean>} - Whether export was successful
 * @private
 */
static async #exportAsSVG(canvas) {
    try {

```

```

    // Get Chart.js instance
    const chartInstance = Chart.getChart(canvas);

    if (!chartInstance) {
        throw new Error('No Chart.js instance found for canvas');
    }

    // Create SVG
    const svgData = chartInstance.toBase64Image('image/svg+xml');

    // Create a temporary Link
    const link = document.createElement('a');

    // Set Link properties
    link.download = `chart-${new Date().toISOString().substring(0, 10)}.svg`;
    link.href = svgData;

    // Click link to trigger download
    document.body.appendChild(link);
    link.click();
    document.body.removeChild(link);

    return true;
} catch (error) {
    console.error('Error exporting as SVG:', error);

    // Fallback to PNG if SVG export fails
    console.log('Falling back to PNG export...');
    return this.#exportAsPNG(canvas);
}

}

/**
 * Export chart as PDF
 * @param {HTMLCanvasElement} canvas - Chart canvas
 * @param {HTMLElement} container - Chart container
 * @returns {Promise<boolean>} - Whether export was successful
 * @private
 */
static async #exportAsPDF(canvas, container) {
    try {
        // Try to import PDF generator
        const PDFGenerator = (await import('./pdf-generator.js')).default;

```

```

    // Get chart details
    const detailsElement = container.nextElementSibling;
    const details = detailsElement ? detailsElement.innerHTML : '';

    // Generate PDF
    const result = await PDFGenerator.generateChartPDF(canvas, details);

    return result.success;
} catch (error) {
    console.error('Error exporting as PDF:', error);

    // Fallback to PNG if PDF export fails
    console.log('Falling back to PNG export...');
    return this.#exportAsPNG(canvas);
}

}

/**
 * Export chart data as Excel
 * @param {HTMLElement} container - Chart container
 * @returns {Promise<boolean>} - Whether export was successful
 * @private
 */
static async #exportAsExcel(container) {
    try {
        // Try to import Excel exporter
        const ExcelExporter = (await import('./excel-exporter.js')).default;

        // Find the closest table in the container or details
        const table = container.querySelector('table') ||
                      container.nextElementSibling?.querySelector('table');

        if (!table) {
            throw new Error('No data table found for Excel export');
        }

        // Extract data from table
        const data = this.#extractTableData(table);

        // Generate Excel file
        const result = await ExcelExporter.generateExcel(data);

        return result.success;
    } catch (error) {

```

```

        console.error('Error exporting as Excel:', error);
        return false;
    }
}

/**
 * Extract data from HTML table
 * @param {HTMLTableElement} table - Table element
 * @returns {Array<Array<string>>} - Table data
 * @private
 */
static #extractTableData(table) {
    const data = [];

    // Extract header row
    const headerRow = table.querySelector('thead tr');
    if (headerRow) {
        const headers = [];
        headerRow.querySelectorAll('th').forEach(cell => {
            headers.push(cell.textContent.trim());
        });
        data.push(headers);
    }

    // Extract data rows
    const rows = table.querySelectorAll('tbody tr');
    rows.forEach(row => {
        const rowData = [];
        row.querySelectorAll('td').forEach(cell => {
            rowData.push(cell.textContent.trim());
        });
        data.push(rowData);
    });

    return data;
}

export default ChartExporter;

```

3.3 combined-view-manager.js

javascript

```

/**
 * Combined View Manager for CVD Risk Toolkit
 *
 * Manages the combined view of Framingham Risk Score and QRISK3,
 * including side-by-side comparison of results and combined
 * treatment recommendations.
 */

import ChartRenderer from './chart-renderer.js';

class CombinedViewManager {
    /**
     * Initialize combined view
     * @returns {Promise<boolean>} - Whether initialization was successful
     */
    static async initialize() {
        try {
            console.log('Initializing combined view...');

            // Setup update button
            this.#setupUpdateButton();

            // Setup export button
            this.#setupExportButton();

            // Try to load saved results
            return this.updateCombinedView();
        } catch (error) {
            console.error('Error initializing combined view:', error);
            return false;
        }
    }

    /**
     * Update combined view with latest results
     * @returns {Promise<boolean>} - Whether update was successful
     */
    static async updateCombinedView() {
        try {
            // Get FRS and QRISK3 results
            const frsResult = this.#getSavedResult('frs');
            const qriskResult = this.#getSavedResult('qrisk');

```

```

        if (!frsResult && !qriskResult) {
            this.#showNoDataMessage();
            return false;
        }

        // Update FRS section
        if (frsResult) {
            this.#updateFRSSection(frsResult);
        }

        // Update QRISK3 section
        if (qriskResult) {
            this.#updateQRISK3Section(qriskResult);
        }

        // Update risk visualization
        if (frsResult || qriskResult) {
            await this.#updateRiskVisualization(frsResult, qriskResult);
        }

        // Update risk factor impact
        if (frsResult || qriskResult) {
            await this.#updateRiskFactorImpact(frsResult, qriskResult);
        }

        // Update recommendations
        if (frsResult || qriskResult) {
            await this.#updateRecommendations(frsResult, qriskResult);
        }

        // Enable export button
        const exportButton = document.getElementById('export-report-btn');
        if (exportButton) {
            exportButton.disabled = false;
        }

        return true;
    } catch (error) {
        console.error('Error updating combined view:', error);

        // Show error message
        this.#showErrorMessage('Failed to update combined view. Please try again.');

        return false;
    }
}

```

```

        }

    }

    /**
     * Set up update button
     * @private
     */
    static #setupUpdateButton() {
        const updateButton = document.getElementById('update-combined-btn');
        if (updateButton) {
            updateButton.addEventListener('click', async () => {
                updateButton.disabled = true;
                await this.updateCombinedView();
                updateButton.disabled = false;
            });
        }

        // Enable button if at least one result is available
        const frsResult = this.#getSavedResult('frs');
        const qriskResult = this.#getSavedResult('qrisk');
        updateButton.disabled = !(frsResult || qriskResult);
    }
}

/**
 * Set up export button
 * @private
 */
static #setupExportButton() {
    const exportButton = document.getElementById('export-report-btn');
    if (exportButton) {
        exportButton.addEventListener('click', async () => {
            try {
                // Import PDF generator
                const PDFGenerator = (await import('../data-management/pdf-generator.js')).PDFGenerator;

                // Export combined view
                PDFGenerator.exportCombinedView();
            } catch (error) {
                console.error('Error exporting combined view:', error);
                this.#showErrorMessage('Failed to export report. Please try again.');
            }
        });
    }

    // Disable button if no results are available
}

```

```

        const frsResult = this.#getSavedResult('frs');
        const qriskResult = this.#getSavedResult('qrisk');
        exportButton.disabled = !(frsResult || qriskResult);
    }
}

/***
 * Update FRS section
 * @param {Object} frsResult - FRS result
 * @private
 */
static #updateFRSSection(frsResult) {
    const percentageElement = document.getElementById('combined-frs-percentage');
    const categoryElement = document.getElementById('combined-frs-category').querySelector(
    const statusElement = document.getElementById('combined-frs-status');

    if (percentageElement) {
        const riskValue = frsResult.modifiedRiskPercent || frsResult.tenYearRiskPercent;
        percentageElement.textContent = `${riskValue}%`;
        percentageElement.className = 'risk-percentage';
        percentageElement.classList.add(`risk-${frsResult.riskCategory}`);
    }

    if (categoryElement) {
        categoryElement.textContent = frsResult.riskCategory.charAt(0).toUpperCase() + frsResult.riskCategory.slice(1);
        categoryElement.className = 'risk-category-value';
        categoryElement.classList.add(`risk-${frsResult.riskCategory}`);
    }

    if (statusElement) {
        statusElement.innerHTML =
            `<span class="status-indicator complete">Calculated on ${new Date(frsResult.calculatedAt)}</span>`;
    }
}

/***
 * Update QRISK3 section
 * @param {Object} qriskResult - QRISK3 result
 * @private
 */
static #updateQRISK3Section(qriskResult) {
    const percentageElement = document.getElementById('combined-qrisk-percentage');
    const categoryElement = document.getElementById('combined-qrisk-category').querySelector(

```

```

const statusElement = document.getElementById('combined-qrisk-status');

if (percentageElement) {
    percentageElement.textContent = `${qriskResult.tenYearRiskPercent}%`;
    percentageElement.className = 'risk-percentage';
    percentageElement.classList.add(`risk-${qriskResult.riskCategory}`);
}

if (categoryElement) {
    categoryElement.textContent = qriskResult.riskCategory.charAt(0).toUpperCase() + qriskResult.riskCategory.slice(1);
    categoryElement.className = 'risk-category-value';
    categoryElement.classList.add(`risk-${qriskResult.riskCategory}`);
}

if (statusElement) {
    statusElement.innerHTML =
        `Calculated on ${new Date(qriskResult.calculatedOn)}`;
}
}

/**
 * Update risk visualization
 * @param {Object} frsResult - FRS result
 * @param {Object} qriskResult - QRISK3 result
 * @returns {Promise<boolean>} - Whether update was successful
 * @private
 */
static async #updateRiskVisualization(frsResult, qriskResult) {
    try {
        const container = document.getElementById('combined-risk-chart');
        if (!container) return false;

        // Clear container
        container.innerHTML = '';

        // Render chart
        const options = {
            type: 'comparison',
            comparisonMethod: 'bar',
            showDifferences: true,
            includeGuidelines: true
        };

```

```

        const result = await ChartRenderer.renderChart(container, options);

        return result.success;
    } catch (error) {
        console.error('Error updating risk visualization:', error);
        return false;
    }
}

/**
 * Update risk factor impact
 * @param {Object} frsResult - FRS result
 * @param {Object} qriskResult - QRISK3 result
 * @returns {Promise<boolean>} - Whether update was successful
 * @private
 */
static async #updateRiskFactorImpact(frsResult, qriskResult) {
    try {
        const container = document.getElementById('risk-factor-impact-chart');
        if (!container) return false;

        // Clear container
        container.innerHTML = '';

        // Determine which calculator to use (prefer the one with higher risk)
        const calculator = this.#determineLeadCalculator(frsResult, qriskResult);

        // Render chart
        const options = {
            type: 'risk-factors',
            calculator,
            factors: {
                age: true,
                bp: true,
                cholesterol: true,
                smoking: true,
                diabetes: true
            },
            analysisMethod: 'individual'
        };

        const result = await ChartRenderer.renderChart(container, options);

        return result.success;
    }
}

```

```

        } catch (error) {
            console.error('Error updating risk factor impact:', error);
            return false;
        }
    }

    /**
     * Update recommendations
     * @param {Object} frsResult - FRS result
     * @param {Object} qriskResult - QRISK3 result
     * @returns {Promise<boolean>} - Whether update was successful
     * @private
     */
    static async #updateRecommendations(frsResult, qriskResult) {
        try {
            const container = document.getElementById('combined-recommendation');
            if (!container) return false;

            // Generate recommendations
            const TreatmentRecommendations = (await import('../calculations/treatment-recommenc

            // Get patient data
            const patientData = frsResult?.parameters || qriskResult?.parameters || null;

            if (!patientData) {
                container.innerHTML = '<p>No patient data available for recommendations.</p>';
                return false;
            }

            // Generate recommendations
            const recommendations = await TreatmentRecommendations.generateRecommendations(
                patientData,
                frsResult,
                qriskResult
            );

            if (recommendations.success) {
                container.innerHTML = recommendations.summary;
                return true;
            } else {
                container.innerHTML = `<p>Failed to generate recommendations: ${recommendations.message}</p>`;
                return false;
            }
        } catch (error) {

```

```
        console.error('Error updating recommendations:', error);
        container.innerHTML = '<p>An error occurred while generating recommendations.</p>';
        return false;
    }
}

/**
 * Show no data message
 * @private
 */
static #showNoDataMessage() {
    // Update FRS section
    const frsStatus = document.getElementById('combined-frs-status');
    if (frsStatus) {
        frsStatus.innerHTML = '<span class="status-indicator incomplete">Not calculated</span>';
    }

    // Update QRISK3 section
    const qriskStatus = document.getElementById('combined-qrisk-status');
    if (qriskStatus) {
        qriskStatus.innerHTML = '<span class="status-indicator incomplete">Not calculated</span>';
    }

    // Update risk chart
    const riskChart = document.getElementById('combined-risk-chart');
    if (riskChart) {
        riskChart.innerHTML =
            `<div class="chart-placeholder">
                <p>No risk calculation data available. Please calculate risk using the Frameworks tab above.
            </div>
`;
    }

    // Update risk factor impact chart
    const impactChart = document.getElementById('risk-factor-impact-chart');
    if (impactChart) {
        impactChart.innerHTML =
            `<div class="chart-placeholder">
                <p>No risk calculation data available. Please calculate risk using the Frameworks tab above.
            </div>
`;
    }

    // Update recommendations
}
```

```

const recommendationContainer = document.getElementById('combined-recommendation');
if (recommendationContainer) {
    recommendationContainer.innerHTML =
        <p>No risk calculation data available. Please calculate risk using the Framing</p>;
}
}

/**
 * Show error message
 * @param {string} message - Error message
 * @private
 */
static #showErrorMessage(message) {
    // Try to use error notification system
    try {
        const ErrorDetectionSystem = (await import('../utils/error-detection-system.js')).ErrorDetectionSystem;
        ErrorDetectionSystem.showErrorNotification(new Error(message), 'Combined View Error');
    } catch (error) {
        // Fallback to alert
        alert(message);
    }
}

/**
 * Get saved result from session storage
 * @param {string} calculator - Calculator type
 * @returns {Object|null} - Saved result or null if not found
 * @private
 */
static #getSavedResult(calculator) {
    try {
        const result = sessionStorage.getItem(` ${calculator}_result`);

        if (result) {
            return JSON.parse(result);
        }
    } catch (error) {
        console.error(`Error retrieving ${calculator} result from session storage:`, error);
    }

    return null;
}

```

```

/**
 * Determine lead calculator
 * @param {Object} frsResult - FRS result
 * @param {Object} qriskResult - QRISK3 result
 * @returns {string} - Lead calculator
 * @private
 */
static #determineLeadCalculator(frsResult, qriskResult) {
    if (frsResult && !qriskResult) {
        return 'frs';
    }

    if (!frsResult && qriskResult) {
        return 'qrisk';
    }

    if (frsResult && qriskResult) {
        const frsRisk = frsResult.modifiedRiskPercent || frsResult.tenYearRiskPercent;
        const qriskRisk = qriskResult.tenYearRiskPercent;

        return frsRisk >= qriskRisk ? 'frs' : 'qrisk';
    }
}

return 'frs'; // Default to FRS
}
}

export default CombinedViewManager;
```
 <div class="risk-summary">
 <p>Framingham Risk Score: ${frsRisk.toFixed(1)}% (${frsCategory})
 <p>QRISK3: ${qriskRisk.toFixed(1)}% (${qriskCategory} risk)</p>
 ${showDifferences ? `<p>Difference: ${riskDifference}% (${relativeRisk})</p>` : ''}
 </div>
 ${includeGuidelines ? `
 <div class="guidelines-summary">
 <h4>Guideline Recommendations</h4>
 <table>
 <thead>
 <tr>
 <th>Guideline</th>
 <th>Framingham</th>
 <th>QRISK3</th>
 </tr>
 </thead>
 <tbody>
 <tr>
 <td>${guideline}</td>
 <td>${frsRisk}</td>
 <td>${qriskRisk}</td>
 </tr>
 </tbody>
 </table>
 </div>
 ` : ''}

```

```

 <tbody>
 <tr>
 <td>Canadian (CCS 2021)</td>
 <td>${this.#getGuidelineRecommendation('ccs2021', frsCategory)}</td>
 <td>${this.#getGuidelineRecommendation('ccs2021', qriskCategory)}</td>
 </tr>
 </tbody>
 </table>
</div>
` : ''}
<div class="chart-interpretation">
 <h4>Interpretation</h4>
 <p>The Framingham Risk Score and QRISK3 calculators use different algorithms and approaches to calculate risk. Some key differences include:</p>

 Different study populations used to develop the algorithms
 Different risk factors included in the calculations
 Different statistical methods used to derive the formulas

 <p>Clinical guidelines generally recommend using the higher risk estimate for treatment decisions.</p>
</div>
</div>`;
}

return { chart, details };
}

/**
 * Create comparison radar chart
 * @param {HTMLElement} container - Chart container
 * @param {Object} patientData - Patient data
 * @param {number} frsRisk - FRS risk
 * @param {number} qriskRisk - QRISK3 risk
 * @param {string} frsCategory - FRS category
 * @param {string} qriskCategory - QRISK3 category
 * @param {Object} colors - Color scheme
 * @returns {Promise<Object>} - Chart and details
 * @private
 */
static async #createComparisonRadarChart(
 container,
 patientData,
 frsRisk,
 qriskRisk,
 frsCategory,
 qriskCategory,

```

```

colors
) {
 // Create chart
 const chartCanvas = this.#createCanvasElement(container);
 const chartContext = chartCanvas.getContext('2d');

 // Normalize factors for comparison
 const factors = [
 'Age',
 'Blood Pressure',
 'Cholesterol',
 'Smoking',
 'Diabetes',
 'Family History'
];

 // Define importance of each factor in each calculator (0-1 scale)
 const factorImportance = {
 frs: {
 'Age': 0.9,
 'Blood Pressure': 0.8,
 'Cholesterol': 0.8,
 'Smoking': 0.7,
 'Diabetes': 0.6,
 'Family History': 0.3 // FRS doesn't directly include family history
 },
 qrisk: {
 'Age': 0.9,
 'Blood Pressure': 0.7,
 'Cholesterol': 0.6,
 'Smoking': 0.7,
 'Diabetes': 0.8,
 'Family History': 0.7
 }
 };

 // Create radar chart
 const chart = new Chart(chartContext, {
 type: 'radar',
 data: {
 labels: factors,
 datasets: [
 {
 label: 'Framingham Risk Score',

```

```
 data: factors.map(factor => factorImportance.frs[factor] * 100),
 backgroundColor: colors.background.primary,
 borderColor: colors.primary,
 borderWidth: 1,
 pointBackgroundColor: colors.primary
 },
{
 label: 'QRISK3',
 data: factors.map(factor => factorImportance.qrisk[factor] * 100),
 backgroundColor: colors.background.secondary,
 borderColor: colors.secondary,
 borderWidth: 1,
 pointBackgroundColor: colors.secondary
}
]
},
options: {
 responsive: true,
 maintainAspectRatio: false,
 scales: {
 r: {
 beginAtZero: true,
 max: 100,
 ticks: {
 display: false
 }
 }
 },
 plugins: {
 title: {
 display: true,
 text: 'Risk Factor Weightings Comparison'
 },
 tooltip: {
 callbacks: {
 label: function(context) {
 const dataset = context.dataset.label;
 const value = context.parsed.r;
 return `${dataset}: ${Math.round(value)}% weighting`;
 }
 }
 }
 }
}
```

```

});;

// Calculate risk difference
const riskDifference = Math.abs(frsRisk - qriskRisk).toFixed(1);
const relativeDifference = Math.round((Math.abs(frsRisk - qriskRisk) / ((frsRisk + qriskRisk) / 2)) * 100);

// Create details HTML
const details = `<div class="chart-details-container">
 <h3>Risk Calculator Comparison</h3>
 <p>This radar chart compares how the Framingham Risk Score and QRISK3 calculators weight different risk factors. The chart shows the absolute difference between the two scores and the percentage relative difference. The legend indicates which calculator assigned more weight to each factor: Framingham Risk Score (blue), QRISK3 (orange), or neither (green).</p>
 <div class="risk-summary">
 <p>Framingham Risk Score: ${frsRisk.toFixed(1)}% (${frsCategory} risk)</p>
 <p>QRISK3: ${qriskRisk.toFixed(1)}% (${qriskCategory} risk)</p>
 <p>Difference: ${riskDifference}% (${relativeDifference}% relative difference)</p>
 </div>
 <div class="calculator-differences">
 <h4>Key Differences Between Calculators</h4>
 <table>
 <thead>
 <tr>
 <th>Feature</th>
 <th>Framingham Risk Score</th>
 <th>QRISK3</th>
 </tr>
 </thead>
 <tbody>
 <tr>
 <td>Development Population</td>
 <td>US (Framingham, MA)</td>
 <td>UK</td>
 </tr>
 <tr>
 <td>Publication Year</td>
 <td>2008</td>
 <td>2017</td>
 </tr>
 <tr>
 <td>Age Range</td>
 <td>30-79</td>
 <td>25-84</td>
 </tr>
 <tr>
 <td>Additional Risk Factors</td>
 <td>Limited</td>
 </tr>
 </tbody>
 </table>
 </div>
</div>`;

```

```

 <td>Extensive (includes ethnicity, RA, etc.)</td>
 </tr>
 </tbody>
 </div>
 <div class="chart-interpretation">
 <h4>Interpretation</h4>
 <p>The radar chart shows the relative emphasis that each calculator places on various risk factors. Clinical guidelines generally recommend using the higher risk estimate for treatment decisions.</p>
 <p>Clinical guidelines generally recommend using the higher risk estimate for treatment decisions.</p>
 </div>
</div>`;

return { chart, details };
}

/**
 * Create comparison table
 * @param {HTMLElement} container - Chart container
 * @param {Object} patientData - Patient data
 * @param {number} frsRisk - FRS risk
 * @param {number} qriskRisk - QRISK3 risk
 * @param {string} frsCategory - FRS category
 * @param {string} qriskCategory - QRISK3 category
 * @param {boolean} includeGuidelines - Whether to include guidelines
 * @returns {Promise<Object>} - Chart and details
 * @private
 */
static async #createComparisonTable(
 container,
 patientData,
 frsRisk,
 qriskRisk,
 frsCategory,
 qriskCategory,
 includeGuidelines
) {
 // Calculate risk difference
 const riskDifference = Math.abs(frsRisk - qriskRisk).toFixed(1);
 const relativeDifference = Math.round((Math.abs(frsRisk - qriskRisk) / ((frsRisk + qriskRisk) / 2)) * 100) / 100;

 // Create a table comparison instead of a chart
 container.innerHTML =
 `<div class="detailed-comparison-table">
 <h3>Detailed Risk Calculator Comparison</h3>

```

```
<table>
 <thead>
 <tr>
 <th>Feature</th>
 <th>Framingham Risk Score</th>
 <th>QRISK3</th>
 </tr>
 </thead>
 <tbody>
 <tr class="highlight-row">
 <td>10-Year Risk</td>
 <td>${frsRisk.toFixed(1)}%</td>
 <td>${qriskRisk.toFixed(1)}%</td>
 </tr>
 <tr>
 <td>Risk Category</td>
 <td>${frsCategory.charAt(0).toUpperCase() + frsCategory.slice(1)}

 <td>${qriskCategory.charAt(0).toUpperCase() + qriskCategory.slice(1)}
```

```

 <tr>
 <td>Family History</td>
 <td>Yes (as modifier)</td>
 <td>Yes (in algorithm)</td>
 </tr>
 <tr>
 <td>Advanced Clinical Factors</td>
 <td>Limited</td>
 <td>Extensive (RA, AF, CKD, etc.)</td>
 </tr>
 ${includeGuidelines ? `

 <tr>
 <td>CCS 2021 Guideline</td>
 <td>${this.#getGuidelineRecommendation('ccs2021', frsCategory)}</td>
 <td>${this.#getGuidelineRecommendation('ccs2021', qriskCategory)}</td>
 </tr>
` : ''}
 </tbody>
</table>
</div>
`;

// Create details HTML
const details = `<div class="chart-details-container">
 <h3>Risk Calculator Comparison</h3>
 <p>This detailed table compares the Framingham Risk Score and QRISK3 calculators across different risk factors and clinical factors.</p>
 <div class="risk-summary">
 <p>Framingham Risk Score: ${frsRisk.toFixed(1)}% (${frsCategory})</p>
 <p>QRISK3: ${qriskRisk.toFixed(1)}% (${qriskCategory} risk)</p>
 <p>Difference: ${riskDifference}% (${relativeDifference} relative difference)</p>
 </div>
 <div class="calculator-notes">
 <h4>Notable Differences</h4>

 Algorithm Age: QRISK3 is more recent (2017) compared to FRS (2013).
 Population: FRS was developed in a US population, while QRISK3 is international.
 Risk Factors: QRISK3 includes more risk factors, such as smoking, alcohol consumption, and family history, while FRS includes fewer factors like systolic blood pressure and diabetes.
 Age Range: QRISK3 covers a wider age range (25-84) compared to FRS (35-74).

 </div>
 <div class="chart-interpretation">
 <h4>Clinical Interpretation</h4>
 <p>Both calculators are validated tools for cardiovascular risk assessment, but their results should be interpreted in context of individual patient characteristics and clinical judgment. Canadian guidelines suggest using FRS as the primary calculator, while UK guidelines suggest using QRISK3 as the primary calculator, with FRS as a secondary option.</p>
 </div>
</div>
`;
```

```

 </div>
 </div>`;

 // No actual chart is created, so return null for chart
 return { chart: null, details };
}

/**
 * Create sensitivity chart
 * @param {HTMLDivElement} container - Chart container
 * @param {Object} options - Chart options
 * @returns {Promise<Object>} - Chart and details
 * @private
 */
static async #createSensitivityChart(container, options) {
 // Get calculator
 const calculator = options.calculator || 'frs';

 // Get patient data from latest risk assessment
 const patientData = await this.#getPatientData();

 if (!patientData) {
 throw new Error('No patient data available. Please calculate risk first.');
 }

 // Set color scheme
 const colorScheme = options.colorScheme || 'default';
 const colors = this.#chartConfig.colors[colorScheme];

 // Get sensitivity parameters
 const paramName = options.sensitivityParam || 'age';
 const rangePct = parseInt(options.sensitivityRange) || 20;
 const steps = parseInt(options.sensitivitySteps) || 5;

 // Map parameter name to actual field name
 const paramMap = {
 'age': 'age',
 'sbp': 'systolicBP',
 'ldl': 'ldl',
 'hdl': 'hdl',
 'tc': 'totalCholesterol'
 };

 // Map parameter name to display name

```

```

const paramDisplayMap = {
 'age': 'Age (years)',
 'sbp': 'Systolic BP (mmHg)',
 'ldl': 'LDL Cholesterol (mmol/L)',
 'hdl': 'HDL Cholesterol (mmol/L)',
 'tc': 'Total Cholesterol (mmol/L)'
};

// Get actual field name
const fieldName = paramMap[paramName];

if (!fieldName) {
 throw new Error(`Unknown parameter: ${paramName}`);
}

if (!patientData[fieldName]) {
 throw new Error(`Parameter ${paramName} not available in patient data`);
}

// Get baseline value
const baselineValue = patientData[fieldName];

// Calculate range
const range = baselineValue * (rangePct / 100);
const minValue = baselineValue - range;
const maxValue = baselineValue + range;

// Generate data points
const dataPoints = [];

// Calculate baseline risk
const baselineRisk = await this.#getBaselineRisk(calculator, patientData);
dataPoints.push({
 value: baselineValue,
 risk: baselineRisk
});

// Calculate step size
const stepSize = (2 * range) / (steps - 1);

// Calculate risk for each point
for (let i = 0; i < steps; i++) {
 const value = minValue + (i * stepSize);
}

```

```

 // Skip if this is the baseline value (already calculated)
 if (Math.abs(value - baselineValue) < 0.001) {
 continue;
 }

 // Create modified data
 const modifiedData = { ...patientData };
 modifiedData[fieldName] = value;

 // Calculate risk
 const risk = await this.#calculateModifiedRisk(calculator, modifiedData);

 dataPoints.push({
 value,
 risk
 });
}

// Sort by value
dataPoints.sort((a, b) => a.value - b.value);

// Find baseline index
const baselineIndex = dataPoints.findIndex(point =>
 Math.abs(point.value - baselineValue) < 0.001
);

// Create chart
const chartCanvas = this.#createCanvasElement(container);
const chartContext = chartCanvas.getContext('2d');

const chart = new Chart(chartContext, {
 type: 'line',
 data: {
 labels: dataPoints.map(point => point.value.toFixed(1)),
 datasets: [
 {
 label: '10-Year CVD Risk (%)',
 data: dataPoints.map(point => point.risk.toFixed(1)),
 borderColor: colors.primary,
 backgroundColor: colors.background.primary,
 fill: false,
 tension: 0.1,
 pointBackgroundColor: dataPoints.map((point, index) =>
 index === baselineIndex ? colors.secondary : colors.primary
)
 }
]
 }
});

```

```
),
 pointRadius: dataPoints.map((point, index) =>
 index === baselineIndex ? 6 : 4
)
 }
]
},
options: {
 responsive: true,
 maintainAspectRatio: false,
 scales: {
 y: {
 beginAtZero: true,
 title: {
 display: true,
 text: '10-Year CVD Risk (%)'
 }
 },
 x: {
 title: {
 display: true,
 text: paramDisplayMap[paramName]
 }
 }
 },
 plugins: {
 title: {
 display: true,
 text: `Risk Sensitivity to ${paramDisplayMap[paramName]}`

 },
 tooltip: {
 callbacks: {
 label: function(context) {
 const index = context.dataIndex;
 const value = dataPoints[index].value;
 const risk = dataPoints[index].risk;
 return `${paramDisplayMap[paramName].split(' ')[0]}: ${value.tc
 },
 afterLabel: function(context) {
 const index = context.dataIndex;
 if (index === baselineIndex) {
 return 'Current value';
 }
 }
 }
 }
 }
}
```

```
 const change = dataPoints[index].risk - baselineRisk;
 return `Change from baseline: ${change > 0 ? '+' : ''}${change}.
 }
},
},
annotation: {
 annotations: {
 line1: {
 type: 'line',
 yMin: 10,
 yMax: 10,
 borderColor: this.#chartConfig.riskCategories.intermediate.colc
 borderWidth: 1,
 borderDash: [4, 4],
 label: {
 content: 'Intermediate Risk',
 enabled: true,
 position: 'left'
 }
 },
 line2: {
 type: 'line',
 yMin: 20,
 yMax: 20,
 borderColor: this.#chartConfig.riskCategories.high.color,
 borderWidth: 1,
 borderDash: [4, 4],
 label: {
 content: 'High Risk',
 enabled: true,
 position: 'left'
 }
 },
 line3: {
 type: 'line',
 xMin: baselineValue,
 xMax: baselineValue,
 yMin: 0,
 yMax: baselineRisk,
 borderColor: colors.secondary,
 borderWidth: 1,
 borderDash: [4, 4]
 }
 }
}
```

```

 }
 }
}

// Calculate slope (risk change per unit change in parameter)
const firstPoint = dataPoints[0];
const lastPoint = dataPoints[dataPoints.length - 1];
const slope = (lastPoint.risk - firstPoint.risk) / (lastPoint.value - firstPoint.value)

// Calculate elasticity (percentage change in risk per percentage change in parameter)
const elasticity = (slope * baselineValue) / baselineRisk;

// Create details HTML
const details = `<div class="chart-details-container">
 <h3>Risk Sensitivity Analysis</h3>
 <p>This chart shows how your cardiovascular risk changes with different values of ${{paramDisplayMap[paramName]}}. The current risk is ${baselineRisk}%.</p>
 <div class="risk-summary">
 <p>Current ${paramDisplayMap[paramName]}: ${firstPoint.value} ${baselineRisk}%.</p>
 <p>Current 10-Year Risk: ${lastPoint.risk}%. (${lastPoint.risk - firstPoint.risk} % change)</p>
 </div>
 <div class="sensitivity-analysis">
 <h4>Sensitivity Analysis</h4>
 <table>
 <thead>
 <tr>
 <th>${paramDisplayMap[paramName]}</th>
 <th>Risk</th>
 <th>Change</th>
 </tr>
 </thead>
 <tbody>
 ${dataPoints.map((point, index) => `
 <tr ${index === baselineIndex ? 'class="highlight-row"' : ''}>
 <td>${point.value}</td>
 <td>${point.risk}</td>
 <td>${index === baselineIndex ? '-' : `${point.risk} > baseline`}</td>
 </tr>
 `).join('')}
 </tbody>
 </table>
 </div>
 <div class="chart-interpretation">
 <h4>Interpretation</h4>

```

```

 <p>Average slope: ${Math.abs(slope).toFixed(2)} risk percentage points
 <p>Elasticity: ${Math.abs(elasticity).toFixed(2)} (meaning a 1% change in
 <p>This analysis shows how your cardiovascular risk is sensitive to changes in
 </div>
</div>`;

return { chart, details };
}

/**
 * Get baseline risk
 * @param {string} calculator - Calculator type
 * @param {Object} patientData - Patient data
 * @returns {Promise<number>} - Baseline risk
 * @private
 */
static async #getBaselineRisk(calculator, patientData) {
 return this.#calculateModifiedRisk(calculator, patientData);
}

/**
 * Calculate modified risk
 * @param {string} calculator - Calculator type
 * @param {Object} patientData - Patient data
 * @returns {Promise<number>} - Modified risk
 * @private
 */
static async #calculateModifiedRisk(calculator, patientData) {
 try {
 // Import calculator module
 let module;

 if (calculator === 'frs') {
 module = await import('./calculations/framingham-algorithm.js');
 } else if (calculator === 'qrisk') {
 module = await import('./calculations/qrisk3-algorithm.js');
 } else {
 throw new Error(`Unknown calculator type: ${calculator}`);
 }

 const Calculator = module.default;

 // Calculate risk
 const result = Calculator.calculateRisk(patientData);
 }
}

```

```

 if (result.success) {
 // Return modified risk if available, otherwise base risk
 return result.modifiedRiskPercent || result.tenYearRiskPercent;
 } else {
 console.error(`Risk calculation failed for ${calculator}:`, result.error);
 return 0;
 }
 } catch (error) {
 console.error(`Error calculating ${calculator} risk:`, error);
 return 0;
 }
}

/**
 * Calculate risk progression
 * @param {string} calculator - Calculator type
 * @param {Object} patientData - Patient data
 * @param {Array<number>} agePoints - Age points
 * @param {boolean} baseline - Whether this is baseline calculation
 * @param {string} intervention - Intervention type
 * @returns {Promise<Array<Object>>} - Risk progression data
 * @private
 */
static async #calculateRiskProgression(
 calculator,
 patientData,
 agePoints,
 baseline = true,
 intervention = 'none'
) {
 const riskPoints = [];

 // Apply intervention if not baseline
 let modifiedData = { ...patientData };

 if (!baseline) {
 switch (intervention) {
 case 'statin':
 // Simulate moderate intensity statin effect
 if (modifiedData.ldl) {
 modifiedData.ldl = modifiedData.ldl * 0.5; // 50% reduction
 }
 }
 }
}

```

```

 if (modifiedData.totalCholesterol && modifiedData.hdl) {
 // Adjust total cholesterol based on LDL reduction
 const nonHdl = modifiedData.totalCholesterol - modifiedData.hdl;
 const reducedNonHdl = nonHdl * 0.5;
 modifiedData.totalCholesterol = reducedNonHdl + modifiedData.hdl;
 }
 break;

 case 'bp-reduction':
 // Simulate BP treatment effect
 modifiedData.systolicBP = Math.max(120, modifiedData.systolicBP - 20);
 modifiedData.onBPMeds = true;
 break;

 case 'smoking-cessation':
 // Simulate smoking cessation
 if (modifiedData.isSmoker) {
 modifiedData.isSmoker = false;

 if (modifiedData.smokingStatus) {
 modifiedData.smokingStatus = 'non';
 }
 }
 break;

 case 'combined':
 // Apply all interventions
 // Statin effect
 if (modifiedData.ldl) {
 modifiedData.ldl = modifiedData.ldl * 0.5;
 }

 if (modifiedData.totalCholesterol && modifiedData.hdl) {
 const nonHdl = modifiedData.totalCholesterol - modifiedData.hdl;
 const reducedNonHdl = nonHdl * 0.5;
 modifiedData.totalCholesterol = reducedNonHdl + modifiedData.hdl;
 }

 // BP treatment
 modifiedData.systolicBP = Math.max(120, modifiedData.systolicBP - 20);
 modifiedData.onBPMeds = true;

 // Smoking cessation
 if (modifiedData.isSmoker) {

```

```

 modifiedData.isSmoker = false;

 if (modifiedData.smokingStatus) {
 modifiedData.smokingStatus = 'non';
 }
 }
 break;
}

}

// Calculate risk for each age point
for (const age of agePoints) {
 const ageData = { ...modifiedData, age };
 const risk = await this.#calculateModifiedRisk(calculator, ageData);

 riskPoints.push({
 age,
 risk
 });
}

return riskPoints;
}

/**
 * Get patient data from latest risk assessment
 * @returns {Promise<Object>} - Patient data
 * @private
 */
static async #getPatientData() {
 // Try to get data from session storage
 try {
 const frsResult = sessionStorage.getItem('frs_result');
 const qriskResult = sessionStorage.getItem('qrisk_result');

 if (frsResult) {
 const resultObj = JSON.parse(frsResult);
 return resultObj.parameters || null;
 } else if (qriskResult) {
 const resultObj = JSON.parse(qriskResult);
 return resultObj.parameters || null;
 }
 } catch (error) {
 console.error('Error retrieving patient data:', error);
 }
}

```

```

}

// If no data found, extract from form fields
try {
 const patientData = {};

 // Try to extract from FRS form
 const frsForm = document.getElementById('frs-form');
 if (
 <thead>
 <tr>
 <th>Risk Factor</th>
 <th>Risk Reduction</th>
 <th>Cumulative Risk</th>
 </tr>
 </thead>
 <tbody>
 <tr>
 <td>Current Risk</td>
 <td>-</td>
 <td>${factorImpact[0].risk.toFixed(1)}%</td>
 </tr>
 ${factorImpact.slice(1).map((item, index) => `

 <tr>
 <td>${item.label}</td>
 <td>${item.reduction.toFixed(1)}%</td>
 <td>${item.risk.toFixed(1)}%</td>
 </tr>
 `).join('')}
 </tbody>
 </table>
</div>
<div class="chart-interpretation">
 <h4>Interpretation</h4>
 <p>This analysis shows the potential cumulative impact of modifying multiple risk factors. Note that risk factors may interact, so the combined effect of modifying multiple risk factors may not be simply additive.</p>
</div>`;

 return { chart, details };
}

/**
 * Create tornado chart
 * @param {HTMLElement} container - Chart container

```

```

* @param {string} calculator - Calculator type
* @param {Object} patientData - Patient data
* @param {number} baselineRisk - Baseline risk
* @param {Object} factors - Factors to analyze
* @param {Object} colors - Color scheme
* @returns {Promise<Object>} - Chart and details
* @private
*/
static async #createTornadoChart(container, calculator, patientData, baselineRisk, factors,
 // Create modified data for each factor (both better and worse)
 const factorImpact = [];

 // Analyze age (if selected)
 if (factors.age) {
 const betterData = { ...patientData, age: Math.max(35, patientData.age - 10) };
 const worseData = { ...patientData, age: Math.min(84, patientData.age + 10) };

 const betterRisk = await this.#calculateModifiedRisk(calculator, betterData);
 const worseRisk = await this.#calculateModifiedRisk(calculator, worseData);

 factorImpact.push({
 factor: 'Age',
 label: `Age ${patientData.age - 10} to ${patientData.age + 10}`,
 better: baselineRisk - betterRisk,
 worse: worseRisk - baselineRisk,
 betterRisk: betterRisk,
 worseRisk: worseRisk
 });
 }

 // Analyze blood pressure (if selected)
 if (factors.bp) {
 const betterData = { ...patientData, systolicBP: 120, onBPMeds: false };
 const worseData = { ...patientData, systolicBP: Math.min(180, patientData.systolicBP) };

 const betterRisk = await this.#calculateModifiedRisk(calculator, betterData);
 const worseRisk = await this.#calculateModifiedRisk(calculator, worseData);

 factorImpact.push({
 factor: 'Blood Pressure',
 label: `Systolic BP (120 to ${worseData.systolicBP} mmHg)`,
 better: baselineRisk - betterRisk,
 worse: worseRisk - baselineRisk,
 betterRisk: betterRisk,
 worseRisk: worseRisk
 });
 }
}

```

```

 worseRisk: worseRisk
 });
}

// Analyze cholesterol (if selected)
if (factors.cholesterol && patientData.totalCholesterol && patientData.hdl) {
 const betterData = {
 ...patientData,
 totalCholesterol: 4.0,
 hdl: 1.4,
 ldl: patientData.ldl ? 2.0 : undefined
 };

 const worseData = {
 ...patientData,
 totalCholesterol: Math.min(8.0, patientData.totalCholesterol * 1.3),
 hdl: Math.max(0.7, patientData.hdl * 0.7),
 ldl: patientData.ldl ? Math.min(6.0, patientData.ldl * 1.5) : undefined
 };

 const betterRisk = await this.#calculateModifiedRisk(calculator, betterData);
 const worseRisk = await this.#calculateModifiedRisk(calculator, worseData);

 factorImpact.push({
 factor: 'Cholesterol',
 label: `Lipid Profile (Optimal to Worse)`,
 better: baselineRisk - betterRisk,
 worse: worseRisk - baselineRisk,
 betterRisk: betterRisk,
 worseRisk: worseRisk
 });
}

// Analyze smoking (if selected)
if (factors.smoking) {
 let betterData, worseData;

 if (patientData.isSmoker === true) {
 // Current smoker - better is non-smoker
 betterData = {
 ...patientData,
 isSmoker: false,
 smokingStatus: patientData.smokingStatus ? 'non' : undefined
 };

```

```

 worseData = { ...patientData }; // No change for worse scenario
 } else {
 // Current non-smoker - worse is smoker
 betterData = { ...patientData }; // No change for better scenario
 worseData = {
 ...patientData,
 isSmoker: true,
 smokingStatus: patientData.smokingStatus ? 'moderate' : undefined
 };
 }

 const betterRisk = await this.#calculateModifiedRisk(calculator, betterData);
 const worseRisk = await this.#calculateModifiedRisk(calculator, worseData);

 factorImpact.push({
 factor: 'Smoking',
 label: `Smoking Status (Non-smoker to Smoker)`,
 better: baselineRisk - betterRisk,
 worse: worseRisk - baselineRisk,
 betterRisk: betterRisk,
 worseRisk: worseRisk
 });
}

// Analyze diabetes (if selected)
if (factors.diabetes) {
 let betterData, worseData;

 if (patientData.hasDiabetes === true) {
 // Current diabetic - better is no diabetes (hypothetical)
 betterData = {
 ...patientData,
 hasDiabetes: false,
 diabetesStatus: patientData.diabetesStatus ? 'none' : undefined
 };
 worseData = { ...patientData }; // No change for worse scenario
 } else {
 // Current non-diabetic - worse is having diabetes
 betterData = { ...patientData }; // No change for better scenario
 worseData = {
 ...patientData,
 hasDiabetes: true,
 diabetesStatus: patientData.diabetesStatus ? 'type2' : undefined
 };
 }
}

```

```

 }

 const betterRisk = await this.#calculateModifiedRisk(calculator, betterData);
 const worseRisk = await this.#calculateModifiedRisk(calculator, worseData);

 factorImpact.push({
 factor: 'Diabetes',
 label: `Diabetes Status (No Diabetes to Diabetes)`,
 better: baselineRisk - betterRisk,
 worse: worseRisk - baselineRisk,
 betterRisk: betterRisk,
 worseRisk: worseRisk
 });
}

// Sort by total impact (better + worse)
factorImpact.sort((a, b) => (b.better + b.worse) - (a.better + a.worse));

// Create chart
const chartCanvas = this.#createCanvasElement(container);
const chartContext = chartCanvas.getContext('2d');

const chart = new Chart(chartContext, {
 type: 'bar',
 data: {
 labels: factorImpact.map(item => item.label),
 datasets: [
 {
 label: 'Risk Reduction (Better)',
 data: factorImpact.map(item => item.better.toFixed(1)),
 backgroundColor: colors.tertiary,
 borderColor: colors.tertiary,
 borderWidth: 1
 },
 {
 label: 'Risk Increase (Worse)',
 data: factorImpact.map(item => item.worse.toFixed(1)),
 backgroundColor: colors.secondary,
 borderColor: colors.secondary,
 borderWidth: 1
 }
]
 },
 options: {

```

```
responsive: true,
maintainAspectRatio: false,
indexAxis: 'y',
scales: {
 x: {
 title: {
 display: true,
 text: 'Change in 10-Year CVD Risk (%)'
 }
 }
},
plugins: {
 title: {
 display: true,
 text: 'Risk Factor Sensitivity Analysis'
 },
 tooltip: {
 callbacks: {
 label: function(context) {
 const datasetIndex = context.datasetIndex;
 const index = context.dataIndex;
 const value = context.parsed.x;

 if (datasetIndex === 0) {
 return `Risk reduction: ${value}%`;
 } else {
 return `Risk increase: ${value}%`;
 }
 },
 afterLabel: function(context) {
 const datasetIndex = context.datasetIndex;
 const index = context.dataIndex;

 if (datasetIndex === 0) {
 return `New risk: ${factorImpact[index].betterRisk.toFixed(1)}`;
 } else {
 return `New risk: ${factorImpact[index].worseRisk.toFixed(1)}`;
 }
 }
 }
 }
}
});
```

```

// Create details HTML
const details = `<div class="chart-details-container">
 <h3>Risk Factor Sensitivity Analysis</h3>
 <p>This chart shows how your cardiovascular risk would change with both improvement
 <div class="risk-summary">
 <p>Current 10-Year Risk: ${baselineRisk.toFixed(1)}% (${this.#`${
 riskType}`})</p>
 </div>
 <div class="factor-impact-table">
 <h4>Detailed Sensitivity Analysis</h4>
 <table>
 <thead>
 <tr>
 <th>Risk Factor</th>
 <th>Better Scenario</th>
 <th>Worse Scenario</th>
 </tr>
 </thead>
 <tbody>
 ${factorImpact.map(item => `
 <tr>
 <td>${item.label}</td>
 <td>${item.betterRisk.toFixed(1)}% ${((item.better > 0) ? "-" : "+")}${item.worseRisk.toFixed(1)}% (+${item.worse.toFixed(1)})</td>
 </tr>
 `).join('')}
 </tbody>
 </table>
 </div>
 <div class="chart-interpretation">
 <h4>Interpretation</h4>
 <p>This tornado diagram shows the impact of both improving and worsening each risk factor. Risk factors with the largest bars have the most influence on your cardiovascular risk.
 </div>
</div>`;

return { chart, details };
}

/**
 * Create treatment effect chart
 * @param {HTMLElement} container - Chart container
 * @param {Object} options - Chart options
 * @returns {Promise<Object>} - Chart and details

```

```
* @private
*/
static async #createTreatmentEffectChart(container, options) {
 // Get calculator
 const calculator = options.calculator || 'both';

 // Get patient data from latest risk assessment
 const patientData = await this.#getPatientData();

 if (!patientData) {
 throw new Error('No patient data available. Please calculate risk first.');
 }

 // Set color scheme
 const colorScheme = options.colorScheme || 'default';
 const colors = this.#chartConfig.colors[colorScheme];

 // Get treatment parameters
 const statinIntensity = options.statinIntensity || 'none';
 const bpTreatment = options.bpTreatment || 'none';

 // Calculate LDL reduction based on statin intensity
 let ldlReduction = 0;
 if (options.ldlReduction === 'custom') {
 ldlReduction = parseInt(options.customLdlReduction) / 100;
 } else if (options.ldlReduction) {
 ldlReduction = parseInt(options.ldlReduction) / 100;
 } else {
 switch (statinIntensity) {
 case 'low':
 ldlReduction = 0.3; // 30%
 break;
 case 'moderate':
 ldlReduction = 0.5; // 50%
 break;
 case 'high':
 ldlReduction = 0.6; // 60%
 break;
 }
 }

 // Calculate SBP reduction based on BP treatment
 let sbpReduction = 0;
 if (options.bpReduction === 'custom') {
```

```

 sbpReduction = parseInt(options.customBpReduction);
 } else if (options.bpReduction) {
 sbpReduction = parseInt(options.bpReduction);
 } else {
 switch (bpTreatment) {
 case 'single':
 sbpReduction = 10;
 break;
 case 'dual':
 sbpReduction = 20;
 break;
 case 'triple':
 sbpReduction = 30;
 break;
 }
 }

 // Calculate baseline risk
 const baselineRisk = await this.#getBaselineRisk(calculator, patientData);

 // Create treatments to analyze
 const treatments = [];

 // Add baseline (no treatment)
 treatments.push({
 name: 'Current Risk',
 description: 'No treatment',
 risk: baselineRisk,
 reduction: 0,
 relativeReduction: 0
 });

 // Add statin therapy if selected
 if (statinIntensity !== 'none') {
 const statinData = { ...patientData };

 // Reduce LDL by statin effect
 if (statinData.ldl) {
 statinData.ldl = statinData.ldl * (1 - ldlReduction);
 }

 // Reduce total cholesterol proportionally
 if (statinData.totalCholesterol && statinData.hdl) {
 const nonHdl = statinData.totalCholesterol - statinData.hdl;

```

```

 const reducedNonHdl = nonHdl * (1 - ldlReduction);
 statinData.totalCholesterol = reducedNonHdl + statinData.hdl;
 }

 const statinRisk = await this.#calculateModifiedRisk(calculator, statinData);
 const statinReduction = baselineRisk - statinRisk;
 const statinRelativeReduction = (statinReduction / baselineRisk) * 100;

 treatments.push({
 name: `${statinIntensity.charAt(0).toUpperCase() + statinIntensity.slice(1)} Ir`,
 description: `LDL-C reduced by ${Math.round(ldlReduction * 100)}%`,
 risk: statinRisk,
 reduction: statinReduction,
 relativeReduction: statinRelativeReduction
 });
}

// Add BP treatment if selected
if (bpTreatment !== 'none') {
 const bpData = { ...patientData };

 // Reduce SBP by treatment effect
 bpData.systolicBP = Math.max(100, bpData.systolicBP - sbpReduction);
 bpData.onBPMeds = true;

 const bpRisk = await this.#calculateModifiedRisk(calculator, bpData);
 const bpReduction = baselineRisk - bpRisk;
 const bpRelativeReduction = (bpReduction / baselineRisk) * 100;

 treatments.push({
 name: `${bpTreatment.charAt(0).toUpperCase() + bpTreatment.slice(1)} BP Therapy`,
 description: `SBP reduced by ${sbpReduction} mmHg`,
 risk: bpRisk,
 reduction: bpReduction,
 relativeReduction: bpRelativeReduction
 });
}

// Add combined therapy if both selected
if (statinIntensity !== 'none' && bpTreatment !== 'none') {
 const combinedData = { ...patientData };

 // Apply both effects
 if (combinedData.ldl) {

```

```

 combinedData.ldl = combinedData.ldl * (1 - ldlReduction);
 }

 if (combinedData.totalCholesterol && combinedData.hdl) {
 const nonHdl = combinedData.totalCholesterol - combinedData.hdl;
 const reducedNonHdl = nonHdl * (1 - ldlReduction);
 combinedData.totalCholesterol = reducedNonHdl + combinedData.hdl;
 }

 combinedData.systolicBP = Math.max(100, combinedData.systolicBP - sbpReduction);
 combinedData.onBPMeds = true;

 const combinedRisk = await this.#calculateModifiedRisk(calculator, combinedData);
 const combinedReduction = baselineRisk - combinedRisk;
 const combinedRelativeReduction = (combinedReduction / baselineRisk) * 100;

 treatments.push({
 name: 'Combined Therapy',
 description: `Statin + BP therapy`,
 risk: combinedRisk,
 reduction: combinedReduction,
 relativeReduction: combinedRelativeReduction
 });
}

// Create chart
const chartCanvas = this.#createCanvasElement(container);
const chartContext = chartCanvas.getContext('2d');

// Create multi-bar chart
const chart = new Chart(chartContext, {
 type: 'bar',
 data: {
 labels: treatments.map(item => item.name),
 datasets: [
 {
 label: '10-Year CVD Risk (%)',
 data: treatments.map(item => item.risk.toFixed(1)),
 backgroundColor: treatments.map((item, index) =>
 index === 0 ? colors.secondary : colors.primary
),
 borderColor: treatments.map((item, index) =>
 index === 0 ? colors.secondary : colors.primary
),
 },
],
 }
});

```

```
borderWidth: 1
}
]
},
options: {
 responsive: true,
 maintainAspectRatio: false,
 scales: {
 y: {
 beginAtZero: true,
 title: {
 display: true,
 text: '10-Year CVD Risk (%)'
 }
 }
 },
 plugins: {
 title: {
 display: true,
 text: 'Treatment Effect Projection'
 },
 tooltip: {
 callbacks: {
 label: function(context) {
 const index = context.dataIndex;
 return `Risk: ${treatments[index].risk.toFixed(1)}%`;
 },
 afterLabel: function(context) {
 const index = context.dataIndex;
 if (index === 0) return null;

 return [
 `Absolute reduction: ${treatments[index].reduction.toFixed(1)}%`,
 `Relative reduction: ${Math.round(treatments[index].relativeReduction)}%`
];
 }
 }
 },
 annotation: {
 annotations: {
 line1: {
 type: 'line',
 yMin: 10,
 yMax: 10,
 xMin: 0,
 xMax: 10
 }
 }
 }
 }
}
```

```

 borderColor: this.#chartConfig.riskCategories.intermediate.color,
 borderWidth: 1,
 borderDash: [4, 4],
 label: {
 content: 'Intermediate Risk',
 enabled: true,
 position: 'left'
 }
 },
 line2: {
 type: 'line',
 yMin: 20,
 yMax: 20,
 borderColor: this.#chartConfig.riskCategories.high.color,
 borderWidth: 1,
 borderDash: [4, 4],
 label: {
 content: 'High Risk',
 enabled: true,
 position: 'left'
 }
 }
 }
 }
}

});

// Create details HTML
const details = `<div class="chart-details-container">
 <h3>Treatment Effect Projection</h3>
 <p>This chart shows the projected impact of different treatment strategies on your risk profile over time.</p>
 <div class="risk-summary">
 <p>Current 10-Year Risk: ${baselineRisk.toFixed(1)}% (${this.#chartConfig.riskCategories.intermediate.color} vs ${this.#chartConfig.riskCategories.high.color})</p>
 </div>
 <div class="treatment-effect-table">
 <h4>Detailed Treatment Effects</h4>
 <table>
 <thead>
 <tr>
 <th>Treatment</th>
 <th>Description</th>
 <th>Projected Risk</th>
 <th>Risk Reduction</th>
 </tr>
 </thead>
 <tbody>
 <tr>
 <td>Strategy A</td>
 <td>Low-risk treatment</td>
 <td>10% Risk</td>
 <td>-10% Risk Reduction</td>
 </tr>
 <tr>
 <td>Strategy B</td>
 <td>Medium-risk treatment</td>
 <td>8% Risk</td>
 <td>-2% Risk Reduction</td>
 </tr>
 <tr>
 <td>Strategy C</td>
 <td>High-risk treatment</td>
 <td>5% Risk</td>
 <td>-5% Risk Reduction</td>
 </tr>
 </tbody>
 </table>
 </div>
</div>`;

```

```

// Create details HTML
const details = `<div class="chart-details-container">
 <h3>Treatment Effect Projection</h3>
 <p>This chart shows the projected impact of different treatment strategies on your risk profile over time.</p>
 <div class="risk-summary">
 <p>Current 10-Year Risk: ${baselineRisk.toFixed(1)}% (${this.#chartConfig.riskCategories.intermediate.color} vs ${this.#chartConfig.riskCategories.high.color})</p>
 </div>
 <div class="treatment-effect-table">
 <h4>Detailed Treatment Effects</h4>
 <table>
 <thead>
 <tr>
 <th>Treatment</th>
 <th>Description</th>
 <th>Projected Risk</th>
 <th>Risk Reduction</th>
 </tr>
 </thead>
 <tbody>
 <tr>
 <td>Strategy A</td>
 <td>Low-risk treatment</td>
 <td>10% Risk</td>
 <td>-10% Risk Reduction</td>
 </tr>
 <tr>
 <td>Strategy B</td>
 <td>Medium-risk treatment</td>
 <td>8% Risk</td>
 <td>-2% Risk Reduction</td>
 </tr>
 <tr>
 <td>Strategy C</td>
 <td>High-risk treatment</td>
 <td>5% Risk</td>
 <td>-5% Risk Reduction</td>
 </tr>
 </tbody>
 </table>
 </div>
</div>`;

```

```

 </tr>
 </thead>
 <tbody>
 ${treatments.map((item, index) =>
 <tr>
 <td>${item.name}</td>
 <td>${item.description}</td>
 <td>${item.risk.toFixed(1)}% (${this.#getRiskCategoryLabel(item.risk)})</td>
 <td>${index === 0 ? '-' : `${item.reduction.toFixed(1)}% (${Math.abs(item.reduction)}%)`}</td>
 </tr>
).join('')}
 </tbody>
</table>
</div>
<div class="chart-interpretation">
 <h4>Interpretation</h4>
 <p>This analysis shows the projected impact of different treatment strategies on risk reduction. The chart compares the absolute risk reduction for each strategy, categorized by risk level: Low risk (< 10%), Intermediate risk (10-19.9%), and High risk (>= 20%).</p>

 Low risk: <10%
 Intermediate risk: 10-19.9%
 High risk: >=20%

 <p>Risk reductions are shown as both absolute (percentage point) and relative (percentage of baseline risk).</p>
</div>;
<div>`

 return { chart, details };
</div>
}

/**
 * Create comparison chart
 * @param {HTMLElement} container - Chart container
 * @param {Object} options - Chart options
 * @returns {Promise<Object>} - Chart and details
 * @private
 */
static async #createComparisonChart(container, options) {
 // Get patient data from latest risk assessment
 const patientData = await this.#getPatientData();

 if (!patientData) {
 throw new Error('No patient data available. Please calculate risk first.');
 }
}

```

```
// Set color scheme
const colorScheme = options.colorScheme || 'default';
const colors = this.#chartConfig.colors[colorScheme];

// Get comparison method
const comparisonMethod = options.comparisonMethod || 'bar';

// Show differences
const showDifferences = options.showDifferences !== false;

// Include guidelines
const includeGuidelines = options.includeGuidelines !== false;

// Calculate risk for both calculators
const frsRisk = await this.#calculateModifiedRisk('frs', patientData);
const qriskRisk = await this.#calculateModifiedRisk('qrisk', patientData);

// Get risk categories
const frsCategory = this.#getRiskCategory(frsRisk);
const qriskCategory = this.#getRiskCategory(qriskRisk);

// Create comparison data based on method
if (comparisonMethod === 'bar') {
 return this.#createComparisonBarChart(
 container,
 patientData,
 frsRisk,
 qriskRisk,
 frsCategory,
 qriskCategory,
 showDifferences,
 includeGuidelines,
 colors
);
} else if (comparisonMethod === 'radar') {
 return this.#createComparisonRadarChart(
 container,
 patientData,
 frsRisk,
 qriskRisk,
 frsCategory,
 qriskCategory,
 colors
);
}
```

```

 } else if (comparisonMethod === 'table') {
 return this.#createComparisonTable(
 container,
 patientData,
 frsRisk,
 qriskRisk,
 frsCategory,
 qriskCategory,
 includeGuidelines
);
 } else {
 throw new Error(`Unknown comparison method: ${comparisonMethod}`);
 }
 }

 /**
 * Create comparison bar chart
 * @param {HTMLElement} container - Chart container
 * @param {Object} patientData - Patient data
 * @param {number} frsRisk - FRS risk
 * @param {number} qriskRisk - QRISK3 risk
 * @param {string} frsCategory - FRS category
 * @param {string} qriskCategory - QRISK3 category
 * @param {boolean} showDifferences - Whether to show differences
 * @param {boolean} includeGuidelines - Whether to include guidelines
 * @param {Object} colors - Color scheme
 * @returns {Promise<Object>} - Chart and details
 * @private
 */
 static async #createComparisonBarChart(
 container,
 patientData,
 frsRisk,
 qriskRisk,
 frsCategory,
 qriskCategory,
 showDifferences,
 includeGuidelines,
 colors
) {
 // Create chart
 const chartCanvas = this.#createCanvasElement(container);
 const chartContext = chartCanvas.getContext('2d');

```

```

// Calculate risk difference
const riskDifference = Math.abs(frsRisk - qriskRisk).toFixed(1);
const relativeDifference = Math.round((Math.abs(frsRisk - qriskRisk) / ((frsRisk + qriskRisk) / 2)).toFixed(2)) * 100;

// Create bar chart
const chart = new Chart(chartContext, {
 type: 'bar',
 data: {
 labels: ['Framingham Risk Score', 'QRISK3'],
 datasets: [
 {
 label: '10-Year CVD Risk (%)',
 data: [frsRisk.toFixed(1), qriskRisk.toFixed(1)],
 backgroundColor: [colors.primary, colors.secondary],
 borderColor: [colors.primary, colors.secondary],
 borderWidth: 1
 }
]
 },
 options: {
 responsive: true,
 maintainAspectRatio: false,
 scales: {
 y: {
 beginAtZero: true,
 title: {
 display: true,
 text: '10-Year CVD Risk (%)'
 }
 }
 },
 plugins: {
 title: {
 display: true,
 text: 'Calculator Comparison'
 },
 tooltip: {
 callbacks: {
 label: function(context) {
 const value = context.parsed.y;
 const index = context.dataIndex;
 return `Risk: ${value}% (${index === 0 ? frsCategory : qriskCategory})`;
 }
 }
 }
 }
 }
});

```

```

 },
 annotation: {
 annotations: {
 line1: {
 type: 'line',
 yMin: 10,
 yMax: 10,
 borderColor: this.#chartConfig.riskCategories.intermediate.colc
 borderWidth: 1,
 borderDash: [4, 4],
 label: {
 content: 'Intermediate Risk',
 enabled: true,
 position: 'left'
 }
 },
 line2: {
 type: 'line',
 yMin: 20,
 yMax: 20,
 borderColor: this.#chartConfig.riskCategories.high.color,
 borderWidth: 1,
 borderDash: [4, 4],
 label: {
 content: 'High Risk',
 enabled: true,
 position: 'left'
 }
 }
 }
 }
 }
});

});
```

```

// Create details HTML
const details = `<div class="chart-details-container">
 <h3>Risk Calculator Comparison</h3>
 <p>This chart compares the 10-year cardiovascular risk estimates from two different
 <div class="risk-summary" primary: 'rgba(0, 0, 0, 0.2)', secondary: 'rgba(80, 80, 80, 0.2)', tertiary: 'rgba(160, 160, 160, 0.2)', quaternary: 'rgba(200, 200, 200, 0.2)'>
 </div>
</p>
```

```

 }
 },
 // Risk categories styling
 riskCategories: {
 low: {
 color: 'rgba(75, 192, 192, 1)',
 backgroundColor: 'rgba(75, 192, 192, 0.2)',
 label: 'Low Risk'
 },
 intermediate: {
 color: 'rgba(255, 159, 64, 1)',
 backgroundColor: 'rgba(255, 159, 64, 0.2)',
 label: 'Intermediate Risk'
 },
 high: {
 color: 'rgba(255, 99, 132, 1)',
 backgroundColor: 'rgba(255, 99, 132, 0.2)',
 label: 'High Risk'
 },
 veryHigh: {
 color: 'rgba(153, 0, 0, 1)',
 backgroundColor: 'rgba(153, 0, 0, 0.2)',
 label: 'Very High Risk'
 }
 }
};

/**
 * Create a risk progression chart showing how risk changes over time
 * @param {HTMLElement} container - Chart container
 * @param {Object} options - Chart options
 * @returns {Promise<Object>} - Chart rendering result
 */
static async renderChart(container, options) {
 try {
 console.log('Rendering chart with options:', options);

 if (!container) {
 throw new Error('Chart container is required');
 }

 // Get chart type
 const type = options.type || 'time-series';
 }
}

```

```

// Create chart based on type
let chart = null;
let details = '';

switch (type) {
 case 'time-series':
 const timeSeriesResult = await this.#createTimeSeriesChart(container, optic
 chart = timeSeriesResult.chart;
 details = timeSeriesResult.details;
 break;

 case 'risk-factors':
 const factorsResult = await this.#createRiskFactorsChart(container, options
 chart = factorsResult.chart;
 details = factorsResult.details;
 break;

 case 'treatment-effect':
 const treatmentResult = await this.#createTreatmentEffectChart(container, c
 chart = treatmentResult.chart;
 details = treatmentResult.details;
 break;

 case 'comparison':
 const comparisonResult = await this.#createComparisonChart(container, optic
 chart = comparisonResult.chart;
 details = comparisonResult.details;
 break;

 case 'sensitivity':
 const sensitivityResult = await this.#createSensitivityChart(container, opt
 chart = sensitivityResult.chart;
 details = sensitivityResult.details;
 break;

 default:
 throw new Error(`Unknown chart type: ${type}`);
}

return {
 success: true,
 chart: chart,
 details: details,
 container: container,
}

```

```

 options: options
 };
} catch (error) {
 console.error('Error rendering chart:', error);

 // Display error message in container
 container.innerHTML =
 <div class="chart-error">
 <div class="error-icon">⚠</div>
 <h3>Chart Rendering Error</h3>
 <p>${error.message || 'Unknown error'}</p>
 </div>
 ;
}

return {
 success: false,
 error: error.message || 'Unknown error',
 details: error.stack || 'No error details available'
};
}

/**
 * Create time series chart showing risk progression over time
 * @param {HTMLElement} container - Chart container
 * @param {Object} options - Chart options
 * @returns {Promise<Object>} - Chart and details
 * @private
 */
static async #createTimeSeriesChart(container, options) {
 // Get calculator
 const calculator = options.calculator || 'both';

 // Get patient data from latest risk assessment
 const patientData = await this.#getPatientData();

 if (!patientData) {
 throw new Error('No patient data available. Please calculate risk first.');
 }

 // Get projection parameters
 const projectionYears = parseInt(options.projectionYears) || 10;
 const ageIncrement = parseInt(options.ageIncrement) || 5;
 const includeInterventions = options.includeInterventions || 'none';
}

```

```

// Set color scheme
const colorScheme = options.colorScheme || 'default';
const colors = this.#chartConfig.colors[colorScheme];

// Generate age points for X-axis
const currentAge = patientData.age;
const agePoints = [];
for (let i = 0; i <= projectionYears; i += ageIncrement) {
 agePoints.push(currentAge + i);
}

// Ensure the final projection year is included
if (agePoints[agePoints.length - 1] < currentAge + projectionYears) {
 agePoints.push(currentAge + projectionYears);
}

// Create datasets
const datasets = [];

// FRS baseline risk progression
if (calculator === 'frs' || calculator === 'both') {
 const frsData = await this.#calculateRiskProgression(
 'frs',
 patientData,
 agePoints,
 includeInterventions === 'none'
);

 datasets.push({
 label: 'Framingham Risk Score',
 data: frsData.map(point => point.risk),
 borderColor: colors.primary,
 backgroundColor: colors.background.primary,
 fill: false,
 tension: 0.1
 });
}

// Add intervention line if requested
if (includeInterventions !== 'none') {
 const frsInterventionData = await this.#calculateRiskProgression(
 'frs',
 patientData,
 agePoints,

```

```

 false,
 includeInterventions
);

 datasets.push({
 label: `FRS with ${this.#getInterventionLabel(includeInterventions)}`,
 data: frsInterventionData.map(point => point.risk),
 borderColor: colors.tertiary,
 backgroundColor: colors.background.tertiary,
 borderDash: [5, 5],
 fill: false,
 tension: 0.1
 });
}

}

// QRISK3 baseline risk progression
if (calculator === 'qrisk' || calculator === 'both') {
 const qriskData = await this.#calculateRiskProgression(
 'qrisk',
 patientData,
 agePoints,
 includeInterventions === 'none'
);

 datasets.push({
 label: 'QRISK3',
 data: qriskData.map(point => point.risk),
 borderColor: colors.secondary,
 backgroundColor: colors.background.secondary,
 fill: false,
 tension: 0.1
 });
}

// Add intervention line if requested
if (includeInterventions !== 'none') {
 const qriskInterventionData = await this.#calculateRiskProgression(
 'qrisk',
 patientData,
 agePoints,
 false,
 includeInterventions
);
}

```

```

 datasets.push({
 label: `QRISK3 with ${this.#getInterventionLabel(includeInterventions)}`,
 data: qriskInterventionData.map(point => point.risk),
 borderColor: colors.quaternary,
 backgroundColor: colors.background.quaternary,
 borderDash: [5, 5],
 fill: false,
 tension: 0.1
 });
 }
}

// Create chart
const chartCanvas = this.#createCanvasElement(container);
const chartContext = chartCanvas.getContext('2d');

const chart = new Chart(chartContext, {
 type: 'line',
 data: {
 labels: agePoints.map(age => `Age ${age}`),
 datasets: datasets
 },
 options: {
 responsive: true,
 maintainAspectRatio: false,
 scales: {
 y: {
 beginAtZero: true,
 title: {
 display: true,
 text: '10-Year CVD Risk (%)'
 }
 },
 x: {
 title: {
 display: true,
 text: 'Age'
 }
 }
 },
 plugins: {
 title: {
 display: true,
 text: 'CVD Risk Progression Over Time'
 }
 }
 }
});

```

```
 },
 tooltip: {
 mode: 'index',
 intersect: false,
 callbacks: {
 label: function(context) {
 return `${context.dataset.label}: ${context.parsed.y.toFixed(1)}
 }
 }
 },
 annotation: {
 annotations: {
 line1: {
 type: 'line',
 yMin: 10,
 yMax: 10,
 borderColor: this.#chartConfig.riskCategories.intermediate.colc
 borderWidth: 1,
 borderDash: [4, 4],
 label: {
 content: 'Intermediate Risk',
 enabled: true,
 position: 'left'
 }
 },
 line2: {
 type: 'line',
 yMin: 20,
 yMax: 20,
 borderColor: this.#chartConfig.riskCategories.high.color,
 borderWidth: 1,
 borderDash: [4, 4],
 label: {
 content: 'High Risk',
 enabled: true,
 position: 'left'
 }
 }
 }
 }
 }
});
```

```

// Create details HTML
const details = `<div class="chart-details-container">
 <h3>Risk Progression Analysis</h3>
 <p>This chart shows how cardiovascular risk is projected to change over time, based on current risk factors. ${includeInterventions === 'none' ? <p>The dashed lines show the projected risk without interventions.</p> : ''}
 <div class="risk-summary">
 <p>Current 10-Year Risk: ${datasets[0].data[0].toFixed(1)}% (${{datasets[0].data[0].toFixed(1)}%})
 <p>Projected Risk in ${projectionYears} years: ${datasets[0].data[1].toFixed(1)}% (${{datasets[0].data[1].toFixed(1)}%})
 ${includeInterventions === 'none' ? <p>Projected Risk with ${this.#getProjectedInterventions()} interventions: ${datasets[0].data[2].toFixed(1)}% (${{datasets[0].data[2].toFixed(1)}%})</p> : ''}
 </div>
 <div class="chart-interpretation">
 <h4>Interpretation</h4>
 <p>Risk increases with age, even if other risk factors remain constant. The horizontal axis represents age, and the vertical axis represents risk percentage. The chart shows three main categories of risk:</p>

 Low risk: <math>< 10\%</math>
 Intermediate risk: $10-19.9\%$
 High risk: $\geq 20\%$

 </div>
</div>`;

return { chart, details };
}

/**
 * Create risk factors chart showing impact of different factors
 * @param {HTMLElement} container - Chart container
 * @param {Object} options - Chart options
 * @returns {Promise<Object>} - Chart and details
 * @private
 */
static async #createRiskFactorsChart(container, options) {
 // Get calculator
 const calculator = options.calculator || 'frs';

 // Get patient data from latest risk assessment
 const patientData = await this.#getPatientData();

 if (!patientData) {
 throw new Error('No patient data available. Please calculate risk first.');
 }

 // Get factors to analyze
 const factors = options.factors || [
 {
 factor: 'Age',
 value: patientData.age
 },
 {
 factor: 'Sex',
 value: patientData.sex
 },
 {
 factor: 'Smoking Status',
 value: patientData.smokingStatus
 },
 {
 factor: 'Hypertension',
 value: patientData.hypertension
 },
 {
 factor: 'Diabetes',
 value: patientData.diabetes
 },
 {
 factor: 'Total Cholesterol',
 value: patientData.totalCholesterol
 },
 {
 factor: 'LDL Cholesterol',
 value: patientData.ldlCholesterol
 },
 {
 factor: 'HDL Cholesterol',
 value: patientData.hdlCholesterol
 },
 {
 factor: 'Systolic Blood Pressure',
 value: patientData.systolicBloodPressure
 },
 {
 factor: 'Fasting Blood Glucose',
 value: patientData.fastingBloodGlucose
 }
];
}

```

```

 age: true,
 bp: true,
 cholesterol: true,
 smoking: true,
 diabetes: true
 };

 // Set color scheme
 const colorScheme = options.colorScheme || 'default';
 const colors = this.#chartConfig.colors[colorScheme];

 // Analysis method
 const analysisMethod = options.analysisMethod || 'individual';

 // Calculate baseline risk
 const baselineRisk = await this.#getBaselineRisk(calculator, patientData);

 // Create chart based on analysis method
 if (analysisMethod === 'individual') {
 return this.#createIndividualFactorsChart(container, calculator, patientData, baselineRisk);
 } else if (analysisMethod === 'combined') {
 return this.#createCombinedFactorsChart(container, calculator, patientData, baselineRisk);
 } else if (analysisMethod === 'tornado') {
 return this.#createTornadoChart(container, calculator, patientData, baselineRisk, factors);
 } else {
 throw new Error(`Unknown analysis method: ${analysisMethod}`);
 }
}

/**
 * Create individual factors chart
 * @param {HTMLElement} container - Chart container
 * @param {string} calculator - Calculator type
 * @param {Object} patientData - Patient data
 * @param {number} baselineRisk - Baseline risk
 * @param {Object} factors - Factors to analyze
 * @param {Object} colors - Color scheme
 * @returns {Promise<Object>} - Chart and details
 * @private
 */
static async #createIndividualFactorsChart(container, calculator, patientData, baselineRisk) {
 // Create modified data for each factor
 const optimizedData = { ...patientData };
 const factorImpact = [];
}

```

```

// Optimize age (if selected)
if (factors.age) {
 // Use younger age (10 years younger)
 optimizedData.age = Math.max(35, patientData.age - 10);
 const optimizedRisk = await this.#calculateModifiedRisk(calculator, optimizedData);
 const impact = baselineRisk - optimizedRisk;

 factorImpact.push({
 factor: 'Age',
 label: `Age ${patientData.age} → ${optimizedData.age}` ,
 impact,
 optimizedRisk: optimizedRisk
 });
}

// Reset to original
optimizedData.age = patientData.age;
}

// Optimize blood pressure (if selected)
if (factors.bp) {
 // Set to optimal BP
 optimizedData.systolicBP = 120;
 optimizedData.onBPMeds = false;

 const optimizedRisk = await this.#calculateModifiedRisk(calculator, optimizedData);
 const impact = baselineRisk - optimizedRisk;

 factorImpact.push({
 factor: 'Blood Pressure',
 label: `Systolic BP ${patientData.systolicBP} → 120 mmHg` ,
 impact,
 optimizedRisk: optimizedRisk
 });
}

// Reset to original
optimizedData.systolicBP = patientData.systolicBP;
optimizedData.onBPMeds = patientData.onBPMeds;
}

// Optimize cholesterol (if selected)
if (factors.cholesterol) {
 // Set to optimal cholesterol
 if (optimizedData.totalCholesterol && optimizedData.hdl) {

```

```

const originalTC = optimizedData.totalCholesterol;
const originalHDL = optimizedData.hdl;

optimizedData.totalCholesterol = 4.0;
optimizedData.hdl = 1.4;

if (optimizedData.ldl) {
 optimizedData.ldl = 2.0;
}

const optimizedRisk = await this.#calculateModifiedRisk(calculator, optimizedData);
const impact = baselineRisk - optimizedRisk;

factorImpact.push({
 factor: 'Cholesterol',
 label: `Total Chol/HDL ${originalTC.toFixed(1)}/${originalHDL.toFixed(1)}`,
 impact,
 optimizedRisk: optimizedRisk
});

// Reset to original
optimizedData.totalCholesterol = originalTC;
optimizedData.hdl = originalHDL;

if (optimizedData.ldl) {
 optimizedData.ldl = patientData.ldl;
}
}

// Optimize smoking (if selected)
if (factors.smoking) {
 // Check if patient is a smoker
 if (patientData.isSmoker === true) {
 optimizedData.isSmoker = false;

 if (optimizedData.smokingStatus) {
 optimizedData.smokingStatus = 'non';
 }
 }

 const optimizedRisk = await this.#calculateModifiedRisk(calculator, optimizedData);
 const impact = baselineRisk - optimizedRisk;

 factorImpact.push({

```

```

 factor: 'Smoking',
 label: 'Smoking Cessation',
 impact: impact,
 optimizedRisk: optimizedRisk
 });

 // Reset to original
 optimizedData.isSmoker = patientData.isSmoker;

 if (patientData.smokingStatus) {
 optimizedData.smokingStatus = patientData.smokingStatus;
 }
}

// Optimize diabetes (if selected)
if (factors.diabetes) {
 // Check if patient has diabetes
 if (patientData.hasDiabetes === true) {
 optimizedData.hasDiabetes = false;

 if (optimizedData.diabetesStatus) {
 optimizedData.diabetesStatus = 'none';
 }
 }

 const optimizedRisk = await this.#calculateModifiedRisk(calculator, optimizedData);
 const impact = baselineRisk - optimizedRisk;

 factorImpact.push({
 factor: 'Diabetes',
 label: 'Without Diabetes',
 impact: impact,
 optimizedRisk: optimizedRisk
 });

 // Reset to original
 optimizedData.hasDiabetes = patientData.hasDiabetes;

 if (patientData.diabetesStatus) {
 optimizedData.diabetesStatus = patientData.diabetesStatus;
 }
}
}

```

```
// Sort by impact
factorImpact.sort((a, b) => b.impact - a.impact);

// Create chart
const chartCanvas = this.#createCanvasElement(container);
const chartContext = chartCanvas.getContext('2d');

const chart = new Chart(chartContext, {
 type: 'bar',
 data: {
 labels: factorImpact.map(item => item.label),
 datasets: [
 {
 label: 'Risk Reduction (%)',
 data: factorImpact.map(item => item.impact.toFixed(1)),
 backgroundColor: colors.primary,
 borderColor: colors.primary,
 borderWidth: 1
 }
]
 },
 options: {
 responsive: true,
 maintainAspectRatio: false,
 indexAxis: 'y',
 scales: {
 x: {
 beginAtZero: true,
 title: {
 display: true,
 text: 'Potential Risk Reduction (%)'
 }
 }
 },
 plugins: {
 title: {
 display: true,
 text: 'Risk Factor Impact Analysis'
 },
 tooltip: {
 callbacks: {
 label: function(context) {
 return `Risk reduction: ${context.parsed.x}%`;
 },
 afterLabel: function(context) {
 const index = context.dataIndex;
 }
 }
 }
 }
 }
});
```

```

 return `New risk: ${factorImpact[index].optimizedRisk.toFixed(1)}
 }
}
}
`);

// Create details HTML
const details = `<div class="chart-details-container">
 <h3>Risk Factor Impact Analysis</h3>
 <p>This chart shows the potential impact of modifying individual risk factors on your overall risk profile.</p>
 <div class="risk-summary">
 <p>Current 10-Year Risk: ${baselineRisk.toFixed(1)}% (${this.#getRiskCategoryL
 </div>
 <div class="factor-impact-table">
 <h4>Detailed Impact Analysis</h4>
 <table>
 <thead>
 <tr>
 <th>Risk Factor</th>
 <th>Potential Risk Reduction</th>
 <th>New Risk</th>
 </tr>
 </thead>
 <tbody>
 ${factorImpact.map(item => `
 <tr>
 <td>${item.label}</td>
 <td>${item.impact.toFixed(1)}%</td>
 <td>${item.optimizedRisk.toFixed(1)}% (${this.#getRiskCategoryL
 </tr>
 `).join('')}
 </tbody>
 </table>
 </div>
 <div class="chart-interpretation">
 <h4>Interpretation</h4>
 <p>This analysis shows the potential impact of modifying each risk factor individually.
 <p>Note that risk factors may interact, so the combined effect of modifying multiple factors could be different.
 </div>
</div>`;

return { chart, details };

```

```

}

/**
 * Create combined factors chart
 * @param {HTMLInputElement} container - Chart container
 * @param {string} calculator - Calculator type
 * @param {Object} patientData - Patient data
 * @param {number} baselineRisk - Baseline risk
 * @param {Object} factors - Factors to analyze
 * @param {Object} colors - Color scheme
 * @returns {Promise<Object>} - Chart and details
 * @private
*/
static async #createCombinedFactorsChart(container, calculator, patientData, baselineRisk,
 // Create modified data for each factor
 const optimizedData = { ...patientData };
 const factorImpact = [];

 // Baseline (current) risk
 factorImpact.push({
 label: 'Current Risk',
 risk: baselineRisk
 });

 // Optimize age (if selected)
 if (factors.age) {
 // Use younger age (10 years younger)
 optimizedData.age = Math.max(35, patientData.age - 10);
 const optimizedRisk = await this.#calculateModifiedRisk(calculator, optimizedData);

 factorImpact.push({
 label: '+ Younger Age',
 risk: optimizedRisk,
 reduction: baselineRisk - optimizedRisk
 });
 }

 // Optimize blood pressure (if selected)
 if (factors.bp) {
 // Set to optimal BP
 optimizedData.systolicBP = 120;
 optimizedData.onBPMeds = false;

 const optimizedRisk = await this.#calculateModifiedRisk(calculator, optimizedData);
 }
}

```

```

 const previousRisk = factorImpact[factorImpact.length - 1].risk;

 factorImpact.push({
 label: '+ Optimal BP',
 risk: optimizedRisk,
 reduction: previousRisk - optimizedRisk
 });
}

// Optimize cholesterol (if selected)
if (factors.cholesterol) {
 // Set to optimal cholesterol
 if (optimizedData.totalCholesterol && optimizedData.hdl) {
 optimizedData.totalCholesterol = 4.0;
 optimizedData.hdl = 1.4;

 if (optimizedData.ldl) {
 optimizedData.ldl = 2.0;
 }
 }

 const optimizedRisk = await this.#calculateModifiedRisk(calculator, optimizedData);
 const previousRisk = factorImpact[factorImpact.length - 1].risk;

 factorImpact.push({
 label: '+ Optimal Lipids',
 risk: optimizedRisk,
 reduction: previousRisk - optimizedRisk
 });
}
}

// Optimize smoking (if selected)
if (factors.smoking && patientData.isSmoker === true) {
 optimizedData.isSmoker = false;

 if (optimizedData.smokingStatus) {
 optimizedData.smokingStatus = 'non';
 }
}

const optimizedRisk = await this.#calculateModifiedRisk(calculator, optimizedData);
const previousRisk = factorImpact[factorImpact.length - 1].risk;

factorImpact.push({
 label: '+ No Smoking',

```

```

 risk: optimizedRisk,
 reduction: previousRisk - optimizedRisk
 });
}

// Optimize diabetes (if selected)
if (factors.diabetes && patientData.hasDiabetes === true) {
 optimizedData.hasDiabetes = false;

 if (optimizedData.diabetesStatus) {
 optimizedData.diabetesStatus = 'none';
 }

 const optimizedRisk = await this.#calculateModifiedRisk(calculator, optimizedData);
 const previousRisk = factorImpact[factorImpact.length - 1].risk;

 factorImpact.push({
 label: '+ No Diabetes',
 risk: optimizedRisk,
 reduction: previousRisk - optimizedRisk
 });
}
}

// Create chart
const chartCanvas = this.#createCanvasElement(container);
const chartContext = chartCanvas.getContext('2d');

// Create stacked bar chart
const datasets = [];

// Add the first bar (current risk)
datasets.push({
 label: 'Current Risk',
 data: [factorImpact[0].risk],
 backgroundColor: colors.secondary,
 borderColor: colors.secondary,
 borderWidth: 1
});

// Add reduction bars for each factor
for (let i = 1; i < factorImpact.length; i++) {
 datasets.push({
 label: factorImpact[i].label,
 data: [factorImpact[i].reduction],

```

```
 backgroundColor: colors.primary,
 borderColor: colors.primary,
 borderWidth: 1
 });
}

const chart = new Chart(chartContext, {
 type: 'bar',
 data: {
 labels: ['10-Year CVD Risk'],
 datasets: datasets
 },
 options: {
 responsive: true,
 maintainAspectRatio: false,
 scales: {
 y: {
 stacked: true,
 beginAtZero: true,
 title: {
 display: true,
 text: '10-Year CVD Risk (%)'
 }
 },
 x: {
 stacked: true
 }
 },
 plugins: {
 title: {
 display: true,
 text: 'Combined Risk Factor Impact Analysis'
 },
 tooltip: {
 callbacks: {
 label: function(context) {
 const datasetIndex = context.datasetIndex;
 const dataIndex = context.dataIndex;
 const value = context.raw;

 if (datasetIndex === 0) {
 return `Current risk: ${value.toFixed(1)}%`;
 } else {
 return `${context.dataset.label}: -${value.toFixed(1)}%`;
 }
 }
 }
 }
 }
 }
});
```

```

 }
 }
}
}

});

// Create details HTML
const details = `<div class="chart-details-container">
 <h3>Combined Risk Factor Impact Analysis</h3>
 <p>This chart shows the potential cumulative impact of modifying multiple risk fact
 <div class="risk-summary">
 <p>Current 10-Year Risk: ${baselineRisk.toFixed(1)}% (${this.#
 <p>Optimized Risk: ${factorImpact[factorImpact.length - 1].ris
 <p>Total Risk Reduction: ${((baselineRisk - factorImpact[factor
 </div>
 <div class="factor-impact-table">
 <h4>Detailed Impact Analysis</h4>
 <table>
 <thead>
 <tr>/**
* Treatment Recommendations for CVD Risk Toolkit
*
* Generates evidence-based treatment recommendations based on
* cardiovascular risk assessment results, following the
* 2021 Canadian Cardiovascular Society Guidelines for the
* Management of Dyslipidemia for CVD Prevention.
*
* References:
* Pearson GJ, et al. 2021 Canadian Cardiovascular Society Guidelines
* for the Management of Dyslipidemia for the Prevention of Cardiovascular
* Disease in Adults. Can J Cardiol. 2021;37(8):1129-1150.
*/
}

class TreatmentRecommendations {
 // Guideline thresholds
 static #guidelines = {
 // CCS 2021 Guidelines
 ccs2021: {
 // LDL-C targets by risk category
 ldlTargets: {
 low: '< 2.0 mmol/L or 50% reduction',
 intermediate: '< 2.0 mmol/L or 50% reduction',
 high: '< 1.8 mmol/L or 40% reduction'
 }
 }
 }
}
```

```
 high: '< 1.8 mmol/L or 50% reduction',
 veryHigh: '< 1.4 mmol/L or 50% reduction'
 },
 // Non-HDL-C targets by risk category
 nonHdlTargets: {
 low: '< 2.6 mmol/L',
 intermediate: '< 2.6 mmol/L',
 high: '< 2.4 mmol/L',
 veryHigh: '< 2.0 mmol/L'
 },
 // ApoB targets by risk category
 apoBTargets: {
 low: '< 0.8 g/L',
 intermediate: '< 0.8 g/L',
 high: '< 0.7 g/L',
 veryHigh: '< 0.65 g/L'
 },
 // Statin intensities by risk category
 statinIntensity: {
 low: 'moderate',
 intermediate: 'moderate to high',
 high: 'high',
 veryHigh: 'high'
 },
 // Treatment thresholds
 treatmentThresholds: {
 // Primary prevention
 primary: {
 low: {
 ldl: 5.0, // mmol/L
 nonHdl: 5.8, // mmol/L
 apoB: 1.45, // g/L
 },
 intermediate: {
 ldl: 3.5, // mmol/L
 nonHdl: 4.3, // mmol/L
 apoB: 1.05, // g/L
 },
 high: {
 ldl: 2.0, // mmol/L
 nonHdl: 2.6, // mmol/L
 apoB: 0.8, // g/L
 }
 }
 },
}
```

```

// Secondary prevention (established ASCVD)
secondary: {
 ldl: 1.8, // mmol/L
 nonHdl: 2.4, // mmol/L
 apoB: 0.7 // g/L
}
},
// Statin doses
statins: {
 high: [
 'Atorvastatin 40-80 mg',
 'Rosuvastatin 20-40 mg'
],
 moderate: [
 'Atorvastatin 10-20 mg',
 'Rosuvastatin 5-10 mg',
 'Simvastatin 20-40 mg',
 'Pravastatin 40-80 mg',
 'Lovastatin 40 mg',
 'Fluvastatin XL 80 mg'
],
 low: [
 'Simvastatin 10 mg',
 'Pravastatin 10-20 mg',
 'Lovastatin 20 mg',
 'Fluvastatin 20-40 mg'
]
},
// Add-on therapies
addOnTherapies: {
 ezetimibe: 'Ezetimibe 10 mg',
 pcsk9i: [
 'Evolocumab 140 mg SC every 2 weeks or 420 mg SC monthly',
 'Alirocumab 75-150 mg SC every 2 weeks or 300 mg SC monthly',
 'Inclisiran 284 mg SC at day 1, 90, then every 6 months'
],
 bempedoicAcid: 'Bempedoic acid 180 mg',
 bempedoicAcidEzetimibe: 'Bempedoic acid 180 mg / Ezetimibe 10 mg'
},
// PCSK9i criteria (BC PharmaCare Special Authority)
pcsk9iCriteria: {
 heFH: [
 'Clinical diagnosis of HeFH with genetic confirmation',
 'On maximally tolerated statin + ezetimibe',

```

```

 'LDL-C ≥ 3.6 mmol/L'
],
 ascvd: [
 'Established ASCVD (history of MI, stroke, or symptomatic peripheral arteri
 'On maximally tolerated statin + ezetimibe for at least 90 days',
 'LDL-C ≥ 2.0 mmol/L'
]
}
}

};

// Very high risk criteria
static #veryHighRiskCriteria = [
 'Clinical ASCVD on maximally tolerated statin + ezetimibe with LDL-C > 1.8 mmol/L',
 'Recent ACS (within last 12 months)',
 'Multiple vascular territories affected',
 'ASCVD event while on maximum statin therapy',
 'ASCVD with recurrent events'
];

/***
 * Generate treatment recommendations
 * @param {Object} patientData - Patient data
 * @param {Object} frsResult - Framingham Risk Score result
 * @param {Object} qriskResult - QRISK3 result
 * @returns {Object} - Treatment recommendations
 */
static generateRecommendations(patientData, frsResult, qriskResult) {
 try {
 console.log('Generating treatment recommendations');

 // Validate inputs
 if (!patientData) {
 throw new Error('Patient data is required');
 }

 // Normalize patient data
 const normalizedData = this.#normalizeData(patientData);

 // Determine risk category (use highest risk from either calculator)
 let riskCategory = 'low';
 let riskPercent = 0;
 let calculator = '';

```

```

 if (frsResult && frsResult.success) {
 riskCategory = frsResult.riskCategory;
 riskPercent = frsResult.modifiedRiskPercent || frsResult.tenYearRiskPercent;
 calculator = 'Framingham Risk Score';
 }

 if (qriskResult && qriskResult.success) {
 // Use QRISK3 if it indicates higher risk
 if (this.#getRiskCategoryWeight(qriskResult.riskCategory) > this.#getRiskCategoryWeight(frsResult.riskCategory)) {
 riskCategory = qriskResult.riskCategory;
 riskPercent = qriskResult.tenYearRiskPercent;
 calculator = 'QRISK3';
 }
 }

 // Check for very high risk criteria
 const hasVeryHighRiskCriteria = this.#checkVeryHighRiskCriteria(normalizedData);
 if (hasVeryHighRiskCriteria) {
 riskCategory = 'veryHigh';
 }

 // Determine if primary or secondary prevention
 const isSecondaryPrevention = normalizedData.establishedASCVD === true;
 const preventionType = isSecondaryPrevention ? 'secondary' : 'primary';

 // Get targets from guidelines
 const guideline = 'ccs2021'; // Default to CCS 2021
 const targets = {
 ldl: this.#guidelines[guideline].ldlTargets[riskCategory],
 nonHdl: this.#guidelines[guideline].nonHdlTargets[riskCategory],
 apoB: this.#guidelines[guideline].apoBTargets[riskCategory]
 };

 // Determine if treatment is indicated
 const treatmentIndicated = this.#isTreatmentIndicated(
 normalizedData,
 riskCategory,
 preventionType
);

 // Generate treatment recommendations
 const treatments = this.#getTreatmentRecommendations(
 normalizedData,
 riskCategory,

```

```

 preventionType,
 treatmentIndicated
);

 // Generate lifestyle recommendations
 const lifestyle = this.#getLifestyleRecommendations(normalizedData);

 // Generate monitoring recommendations
 const monitoring = this.#getMonitoringRecommendations(riskCategory, treatments.primaryTreatment);

 // Create summary
 const summary = this.#createSummary(
 riskCategory,
 riskPercent,
 calculator,
 targets,
 treatments,
 lifestyle,
 preventionType
);

 return {
 success: true,
 riskCategory: riskCategory,
 riskPercent: riskPercent,
 calculator: calculator,
 preventionType: preventionType,
 treatmentIndicated: treatmentIndicated,
 ldlTarget: targets.ldl,
 nonHdlTarget: targets.nonHdl,
 apoBTarget: targets.apoB,
 primaryTreatment: treatments.primaryTreatment,
 alternativeTreatments: treatments.alternativeTreatments,
 additionalTreatments: treatments.additionalTreatments,
 pcsk9iEligible: treatments.pcsk9iEligible,
 lifestyle: lifestyle,
 monitoring: monitoring,
 summary: summary,
 guidelineReference: 'Canadian Cardiovascular Society Guidelines (2021)'
 };
} catch (error) {
 console.error('Error generating treatment recommendations:', error);
 return {
 success: false,

```

```

 error: error.message || 'Unknown error',
 message: 'Failed to generate treatment recommendations'
);
}
}

/**
 * Generate recommendations specific to Framingham Risk Score results
 * @param {Object} patientData - Patient data
 * @param {Object} frsResult - Framingham Risk Score result
 * @returns {Object} - FRS-specific recommendations
 */
static generateFRSRecommendations(patientData, frsResult) {
 try {
 if (!frsResult || !frsResult.success) {
 throw new Error('Valid Framingham Risk Score result is required');
 }

 // Generate standard recommendations
 const recommendations = this.generateRecommendations(patientData, frsResult, null);

 // Add FRS-specific information
 const frsSpecific = {
 baseRiskPercent: frsResult.tenYearRiskPercent,
 modifiedRiskPercent: frsResult.modifiedRiskPercent || frsResult.tenYearRiskPercent,
 riskModifiers: frsResult.riskModifiers || []
 };

 // FRS-specific summary
 const summary = `<h3>Framingham Risk Score: ${frsResult.tenYearRiskPercent}% (${frsResult.modifiedRiskPercent ? `<p>Risk after applying modifiers: ${frsResult.modifiedRiskPercent}` : ''})</h3>
<p>Based on the Framingham Risk Score and current guidelines, the following recommendations are provided:</p>

 LDL-C target: ${recommendations.ldlTarget}
 Primary treatment: ${recommendations.primaryTreatment}
 ${recommendations.alternativeTreatments ? `Alternative treatments: ${recommendations.alternativeTreatments}` : ''}

<p>Lifestyle recommendations: ${recommendations.lifestyle.join(', ')}`</p>
<p>Monitoring: ${recommendations.monitoring.join(', ')}`</p>
<p>Based on: Canadian Cardiovascular Society Guidelines (2021)</p>`;

 return {
 ...recommendations,
 frsSpecific,
 };
 }
}

```

```

 summary
 };
}

} catch (error) {
 console.error('Error generating FRS recommendations:', error);
 return {
 success: false,
 error: error.message || 'Unknown error',
 message: 'Failed to generate FRS recommendations'
 };
}

}

/***
 * Generate recommendations specific to QRISK3 results
 * @param {Object} patientData - Patient data
 * @param {Object} qriskResult - QRISK3 result
 * @returns {Object} - QRISK3-specific recommendations
 */
static generateQRISKRecommendations(patientData, qriskResult) {
 try {
 if (!qriskResult || !qriskResult.success) {
 throw new Error('Valid QRISK3 result is required');
 }

 // Generate standard recommendations
 const recommendations = this.generateRecommendations(patientData, null, qriskResult

 // Add QRISK3-specific information
 const qriskSpecific = {
 heartAge: qriskResult.heartAge,
 relativeRisk: qriskResult.relativeRisk
 };

 // QRISK3-specific summary
 const summary = `<h3>QRISK3: ${qriskResult.tenYearRiskPercent}% (${qriskResult.risk
 <p>Heart age: ${qriskResult.heartAge} years</p>
 <p>Relative risk: ${qriskResult.relativeRisk} times higher than person with opt
 <p>Based on the QRISK3 score and current guidelines, the following recommendati

 LDL-C target: ${recommendations.ldlTarget}
 Primary treatment: ${recommendations.primaryTreatment}
 ${recommendations.alternativeTreatments ? `Alternative treatments: ${re

 <p>Lifestyle recommendations: ${recommendations.lifestyle.joir

```

```

<p>Monitoring: ${recommendations.monitoring.join('; ')}</p>
<p>Based on: Canadian Cardiovascular Society Guidelines (2021)</p>`;

return {
 ...recommendations,
 qriskSpecific,
 summary
};
} catch (error) {
 console.error('Error generating QRISK recommendations:', error);
 return {
 success: false,
 error: error.message || 'Unknown error',
 message: 'Failed to generate QRISK recommendations'
 };
}
}

/**
 * Normalize patient data
 * @param {Object} data - Patient data
 * @returns {Object} - Normalized data
 * @private
 */
static #normalizeData(data) {
 const normalizedData = { ...data };

 // Convert string values to appropriate types
 ['ldl', 'nonHdl', 'totalCholesterol', 'hdl', 'apoB', 'lpa', 'bmi', 'systolicBP', 'age']
 .filter(key => key in normalizedData)
 .forEach(key => {
 if (typeof normalizedData[key] === 'string') {
 normalizedData[key] = parseFloat(normalizedData[key]);
 }
 });
}

// Convert string boolean values to actual booleans
[
 'establishedASCVD',
 'heFH',
 'statinIntolerance',
 'onBPMeds',
 'isSmoker',
 'hasDiabetes',
 'familyHistory'
].forEach(key => {

```

```

 if (key in normalizedData) {
 if (normalizedData[key] === 'true') normalizedData[key] = true;
 if (normalizedData[key] === 'false') normalizedData[key] = false;
 }
 });

 return normalizedData;
}

/***
 * Get risk category weight for comparison
 * @param {string} category - Risk category
 * @returns {number} - Weight
 * @private
 */
static #getRiskCategoryWeight(category) {
 const weights = {
 'low': 1,
 'intermediate': 2,
 'high': 3,
 'veryHigh': 4
 };

 return weights[category] || 0;
}

/***
 * Check if very high risk criteria are met
 * @param {Object} data - Normalized patient data
 * @returns {boolean} - Whether very high risk criteria are met
 * @private
 */
static #checkVeryHighRiskCriteria(data) {
 if (data.establishedASCVD) {
 // Recent ACS
 if (data.recentACS) {
 return true;
 }

 // Multiple vascular territories
 if (data.multipleVascularTerritories) {
 return true;
 }
 }
}

```

```

 // ASCVD event while on maximum statin
 if (data.ascvdEventOnStatin) {
 return true;
 }

 // ASCVD with recurrent events
 if (data.recurrentASCVD) {
 return true;
 }

 // On maximally tolerated statin + ezetimibe with LDL-C > 1.8 mmol/L
 if (data.onMaximalTreatment && data.ldl > 1.8) {
 return true;
 }
 }

 return false;
}

/**
 * Determine if lipid-lowering treatment is indicated
 * @param {Object} data - Normalized patient data
 * @param {string} riskCategory - Risk category
 * @param {string} preventionType - Prevention type (primary/secondary)
 * @returns {boolean} - Whether treatment is indicated
 * @private
 */
static #isTreatmentIndicated(data, riskCategory, preventionType) {
 const guideline = 'ccs2021'; // Default to CCS 2021

 // For secondary prevention, treatment is always indicated
 if (preventionType === 'secondary') {
 return true;
 }

 // For primary prevention, check thresholds
 // Certain conditions warrant treatment regardless of lipid levels
 if (data.heFH === true || data.hasDiabetes === true) {
 return true;
 }

 // For other primary prevention, check lipid thresholds by risk category
 const thresholds = this.#guidelines[guideline].treatmentThresholds.primary[riskCategory]

```

```

 if (!thresholds) {
 return false;
 }

 // Check against appropriate thresholds
 if (data.ldl >= thresholds.ldl) {
 return true;
 }

 if (data.nonHdl >= thresholds.nonHdl) {
 return true;
 }

 if (data.apoB >= thresholds.apoB) {
 return true;
 }

 return false;
 }

 /**
 * Get treatment recommendations
 * @param {Object} data - Normalized patient data
 * @param {string} riskCategory - Risk category
 * @param {string} preventionType - Prevention type (primary/secondary)
 * @param {boolean} treatmentIndicated - Whether treatment is indicated
 * @returns {Object} - Treatment recommendations
 * @private
 */
 static #getTreatmentRecommendations(data, riskCategory, preventionType, treatmentIndicated)
 const guideline = 'ccs2021'; // Default to CCS 2021

 // If treatment is not indicated, return lifestyle only
 if (!treatmentIndicated) {
 return {
 primaryTreatment: 'Lifestyle modification',
 alternativeTreatments: null,
 additionalTreatments: null,
 pcsk9iEligible: false
 };
 }

 // Determine statin intensity
 const statinIntensity = this.#guidelines[guideline].statinIntensity[riskCategory];

```

```

// Get statin options
const statinOptions = this.#guidelines[guideline].statins[statinIntensity.split(' ')[0]]

let primaryTreatment = '';
let alternativeTreatments = [];
let additionalTreatments = [];
let pcsk9iEligible = false;

// Check for statin intolerance
if (data.statinIntolerance === true) {
 primaryTreatment = this.#guidelines[guideline].addOnTherapies.ezetimibe;
 alternativeTreatments = [
 this.#guidelines[guideline].addOnTherapies.bempedoicAcid,
 this.#guidelines[guideline].addOnTherapies.bempedoicAcidEzetimibe
];
}

// For very high risk or high risk with ASCVD, consider PCSK9i
if (riskCategory === 'veryHigh' || (riskCategory === 'high' && preventionType === 'primary')) {
 additionalTreatments = this.#guidelines[guideline].addOnTherapies.pcsk9i;
}

// Check PCSK9i eligibility
if (preventionType === 'secondary' && data.ldl >= 2.0) {
 pcsk9iEligible = true;
}

if (data.heFH === true && data.ldl >= 3.6) {
 pcsk9iEligible = true;
}

} else {
 // Standard statin-based therapy
 primaryTreatment = `${statinIntensity} intensity statin (e.g., ${statinOptions[0]})`;

 if (statinOptions.length > 1) {
 alternativeTreatments = statinOptions.slice(1);
 }

 // Add-on therapies for high/very high risk
 if (riskCategory === 'high' || riskCategory === 'veryHigh') {
 additionalTreatments.push(this.#guidelines[guideline].addOnTherapies.ezetimibe)
 }

 // For very high risk or secondary prevention high risk, consider PCSK9i
 if (riskCategory === 'veryHigh' || preventionType === 'secondary') {

```

```

 additionalTreatments.push(this.#guidelines[guideline].addOnTherapies.pcsk9i

 // Check PCSK9i eligibility
 if (preventionType === 'secondary' && data.onMaximalTreatment && data.ldl >
 pcsk9iEligible = true;
 }

 if (data.heFH === true && data.onMaximalTreatment && data.ldl >= 3.6) {
 pcsk9iEligible = true;
 }
 }
}

return {
 primaryTreatment,
 alternativeTreatments: alternativeTreatments.join('; '),
 additionalTreatments: additionalTreatments.join('; '),
 pcsk9iEligible
};
}

/***
 * Get lifestyle recommendations
 * @param {Object} data - Normalized patient data
 * @returns {Array<string>} - Lifestyle recommendations
 * @private
 */
static #getLifestyleRecommendations(data) {
 const recommendations = [
 'Mediterranean diet or low glycemic index diet'
];

 if (data.systolicBP >= 140) {
 recommendations.push('Reduce sodium intake to <2000 mg/day');
 recommendations.push('DASH diet');
 }

 if (data.isSmoker === true) {
 recommendations.push('Smoking cessation');
 }

 if (data.bmi >= 25) {
 recommendations.push('Weight management');
 }
}

```

```

 }

 recommendations.push('Regular physical activity (150+ minutes moderate to vigorous acti

 return recommendations;
}

/***
 * Get monitoring recommendations
 * @param {string} riskCategory - Risk category
 * @param {string} primaryTreatment - Primary treatment
 * @returns {Array<string>} - Monitoring recommendations
 * @private
 */
static #getMonitoringRecommendations(riskCategory, primaryTreatment) {
 const recommendations = [];

 // Lipid monitoring based on treatment intensity
 if (primaryTreatment.includes('statin')) {
 recommendations.push('Lipid panel at baseline, 3 months after treatment initiation,
 recommendations.push('ALT at baseline, 3 months after statin initiation, and if sym
 recommendations.push('CK only if muscle symptoms develop');

 } else {
 recommendations.push('Lipid panel at baseline, 3 months after treatment initiation,
 }

 // More frequent monitoring for high/very high risk
 if (riskCategory === 'high' || riskCategory === 'veryHigh') {
 recommendations.push('Consider more frequent lipid monitoring (every 3-6 months) ur
 }

 return recommendations;
}

/***
 * Create summary text
 * @param {string} riskCategory - Risk category
 * @param {number} riskPercent - Risk percentage
 * @param {string} calculator - Calculator used
 * @param {Object} targets - Lipid targets
 * @param {Object} treatments - Treatment recommendations
 * @param {Array<string>} lifestyle - Lifestyle recommendations
 * @param {string} preventionType - Prevention type (primary/secondary)
 * @returns {string} - Summary HTML
*/

```

```

* @private
*/
static #createSummary(riskCategory, riskPercent, calculator, targets, treatments, lifestyle
 return `<h3>${calculator}: ${riskPercent}% ${preventionType === 'secondary' ? '(seconda
 <p>Based on the current guidelines, the following recommendations apply:</p>
 <h4>Lipid Targets</h4>

 LDL-C target: ${targets.ldl}
 Non-HDL-C target: ${targets.nonHdl}
 ApoB target: ${targets.apoB}

 <h4>Treatment</h4>

 Primary treatment: ${treatments.primaryTreatment}
 ${treatments.alternativeTreatments ? `Alternative treatments: ${treatments.
 ${treatments.additionalTreatments ? `Additional therapies to consider: ${tr
 ${treatments.pcsk9iEligible ? 'Note: May be eligible for F

 <h4>Lifestyle Recommendations</h4>

 ${lifestyle.map(item => `${item}`).join('')}

 <p>Based on: Canadian Cardiovascular Society Guidelines (2021)</p>`;
 }
}

export default TreatmentRecommendations;

```

### 3. Visualization Components

#### 3.1 chart-renderer.js

javascript

```
/**
 * Chart Renderer for CVD Risk Toolkit
 *
 * Renders interactive charts for visualizing cardiovascular risk.
 * Uses Chart.js for rendering, with support for various visualization
 * types including risk progression over time, risk factor impact,
 * treatment effects, and comparison views.
 */

class ChartRenderer {
 // Chart configuration presets
 static #chartConfig = {
 // Color schemes
 colors: {
 default: {
 primary: 'rgba(54, 162, 235, 1)',
 secondary: 'rgba(255, 99, 132, 1)',
 tertiary: 'rgba(75, 192, 192, 1)',
 quaternary: 'rgba(255, 159, 64, 1)',
 background: {
 primary: 'rgba(54, 162, 235, 0.2)',
 secondary: 'rgba(255, 99, 132, 0.2)',
 tertiary: 'rgba(75, 192, 192, 0.2)',
 quaternary: 'rgba(255, 159, 64, 0.2)'
 }
 },
 highcontrast: {
 primary: 'rgba(0, 0, 255, 1)',
 secondary: 'rgba(255, 0, 0, 1)',
 tertiary: 'rgba(0, 128, 0, 1)',
 quaternary: 'rgba(128, 0, 128, 1)',
 background: {
 primary: 'rgba(0, 0, 255, 0.2)',
 secondary: 'rgba(255, 0, 0, 0.2)',
 tertiary: 'rgba(0, 128, 0, 0.2)',
 quaternary: 'rgba(128, 0, 128, 0.2)'
 }
 },
 colorblind: {
 primary: 'rgba(0, 92, 169, 1)',
 secondary: 'rgba(246, 139, 51, 1)',
 tertiary: 'rgba(0, 132, 61, 1)',
 quaternary: 'rgba(166, 9, 61, 1)',
 }
 }
 }
}
```

```

background: {
 primary: 'rgba(0, 92, 169, 0.2)',
 secondary: 'rgba(246, 139, 51, 0.2)',
 tertiary: 'rgba(0, 132, 61, 0.2)',
 quaternary: 'rgba(166, 9, 61, 0.2)'
}
},
monochrome: {
 primary: 'rgba(0, 0, 0, 1)',
 secondary: 'rgba(80, 80, 80, 1)',
 tertiary: 'rgba(160, 160, 160, 1)',
 quaternary: 'rgba(200, 200, 200, 1)',
 background: {
 primary: 'rgba(0, 0, 0, 0.2)',
 secondary: 'rgba(80, 80, 80, 0.2)',### 2.3 treatment-recommendations.js
 }
}

```

```javascript

```

/**
 * Treatment Recommendations for CVD Risk Toolkit
 *
 * Generates evidence-based treatment recommendations based on
 * cardiovascular risk assessment results, following the
 * 2021 Canadian Cardiovascular Society Guidelines for the
 * Management of Dyslipidemia for CVD Prevention.
 *
 * References:
 * Pearson GJ, et al. 2021 Canadian Cardiovascular Society Guidelines
 * for the Management of Dyslipidemia for the Prevention of Cardiovascular
 * Disease in Adults. Can J Cardiol. 2021;37(8):1129-1150.
 */

```

```

class TreatmentRecommendations {
    // Guideline thresholds
    static #guidelines = {
        // CCS 2021 Guidelines
        ccs2021: {
            // LDL-C targets by risk category
            ldlTargets: {
                low: '< 2.0 mmol/L or 50% reduction',
                intermediate: '< 2.0 mmol/L or 50% reduction',
                high: '< 1.8 mmol/L or 50% reduction',
                veryHigh: '< 1.4 mmol/L or 50% reduction'
            },
            // Non-HDL-C targets by risk category
        }
    }
}
```

```

nonHdlTargets: {
    low: '< 2.6 mmol/L',
    intermediate: '< 2.6 mmol/L',
    high: '< 2.4 mmol/L',
    veryHigh: '< 2.0 mmol/L'
},
// A### 2.2 qrisk3-algorithm.js

```
javascript
/**
 * QRISK3 Calculator for CVD Risk Toolkit
 *
 * Implements the 2017 QRISK3 algorithm for estimating
 * 10-year risk of cardiovascular disease.
 *
 * References:
 * Hippisley-Cox J, Coupland C, Brindle P.
 * Development and validation of QRISK3 risk prediction algorithms
 * to estimate future risk of cardiovascular disease: prospective
 * cohort study. BMJ. 2017;357:j2099.
 */

```

```

class QRISK3Calculator {
 // Coefficient tables from the 2017 QRISK3 model
 static #coefficients = {
 // Women's coefficients
 female: {
 // Base coefficients
 age1: 0.0555612344,
 age2: -0.0004658272,
 bmi1: 0.1192938383,
 bmi2: -0.0013294538,
 ethnicOrigin: {
 white: 0,
 indian: 0.1132278158,
 pakistani: 0.2901967046,
 bangladeshi: 0.2939600255,
 other_asian: 0.1080485251,
 black_caribbean: -0.2001819281,
 black_african: -0.3582880202,
 chinese: -0.3877982566,
 other: -0.1007257907
 },
 familyHistoryCHD: 0.4588543021,

```

```
systolicBP: 0.0137853776,
systolicBPVariability: 0.0088745121,
cholesterolRatio: 0.1362481247,
smoking: {
 non: 0,
 ex: 0.2390073537,
 light: 0.5210798951,
 moderate: 0.6529264461,
 heavy: 0.8736402981
},
diabetes: {
 none: 0,
 type1: 1.2343105421,
 type2: 0.8594831513
},
// Additional risk factors
chronicKidneyDisease: 0.5542833716,
atrialFibrillation: 1.5923567474,
migraines: 0.3012672248,
rheumatoidArthritis: 0.3136790605,
systemicLupusErythematosus: 0.7588469742,
severeMentalIllness: 0.1255530391,
bpTreatment: 0.5611671855,
// Interaction terms
age_systolicBP: 0.0001374006,
age_bmi: -0.0006667066,
age_bpTreatment: -0.0050746398,
age_diabetes: {
 none: 0,
 type1: -0.0057129349,
 type2: -0.0046537161
},
age_smoking: {
 non: 0,
 ex: -0.0030558812,
 light: -0.0011647252,
 moderate: -0.0032175832,
 heavy: -0.0062385486
}
},
// Men's coefficients
male: {
 // Base coefficients
 age1: 0.0753358301,
```

```
age2: -0.0005520561,
bmi1: 0.1456724172,
bmi2: -0.0015052768,
ethnicOrigin: {
 white: 0,
 indian: 0.2482856603,
 pakistani: 0.4514083095,
 bangladeshi: 0.4768158869,
 other_asian: 0.0887508020,
 black_caribbean: -0.3925803749,
 black_african: -0.4707165210,
 chinese: -0.2590977447,
 other: -0.0526792816
},
familyHistoryCHD: 0.5405622985,
systolicBP: 0.0125200865,
systolicBPVariability: 0.0079441488,
cholesterolRatio: 0.1533075246,
smoking: {
 non: 0,
 ex: 0.2331894077,
 light: 0.5171929504,
 moderate: 0.6189482511,
 heavy: 0.7932913713
},
diabetes: {
 none: 0,
 type1: 1.2300463097,
 type2: 0.8592181900
},
// Additional risk factors
chronicKidneyDisease: 0.7214909028,
atrialFibrillation: 0.8820923361,
migraines: 0.2885559809,
rheumatoidArthritis: 0.3351057147,
systemicLupusErythematosus: 0.3280613177,
severeMentalIllness: 0.1313891297,
bpTreatment: 0.5103604603,
erectileDysfunction: 0.2304903150,
// Interaction terms
age_systolicBP: 0.0002635962,
age_bmi: -0.0011782144,
age_bpTreatment: -0.0023929099,
age_diabetes: {
```

```

 none: 0,
 type1: -0.0075491274,
 type2: -0.0042569803
 },
 age_smoking: {
 non: 0,
 ex: -0.0015365918,
 light: -0.0028527723,
 moderate: -0.0038762484,
 heavy: -0.0046674230
 }
},
// Survival probability at 10 years
survival: {
 female: 0.987739,
 male: 0.977666
}
};

// Risk categories based on UK guidelines
static #riskCategories = {
 low: { max: 10 },
 intermediate: { min: 10, max: 20 },
 high: { min: 20 }
};

/**
 * Calculate QRISK3 score
 * @param {Object} params - Risk parameters
 * @returns {Object} - Risk calculation results
 */
static calculateRisk(params) {
 try {
 console.log('Calculating QRISK3 with parameters:', params);

 // Validate required parameters
 if (!this.#validateParameters(params)) {
 return {
 success: false,
 error: 'Missing or invalid required parameters',
 missingParams: this.#getMissingParameters(params),
 algorithm: 'qrisk3'
 };
 }
 }
}
```

```

// Normalize parameters
const normalizedParams = this.#normalizeParameters(params);
const sex = normalizedParams.sex;

// Calculate the risk score

// 1. Apply coefficients to each predictor
let sum = 0;

// Age terms (fractional polynomial terms)
const age = normalizedParams.age;
sum += this.#coefficients[sex].age1 * age;
sum += this.#coefficients[sex].age2 * age * age;

// BMI terms (fractional polynomial terms)
const bmi = normalizedParams.bmi;
if (bmi > 0) {
 sum += this.#coefficients[sex].bmi1 * bmi;
 sum += this.#coefficients[sex].bmi2 * bmi * bmi;
}

// Ethnicity
const ethnicity = normalizedParams.ethnicity || 'white';
sum += this.#coefficients[sex].ethnicOrigin[ethnicity];

// Family history of CHD
if (normalizedParams.familyHistory) {
 sum += this.#coefficients[sex].familyHistoryCHD;
}

// Systolic blood pressure
const sbp = normalizedParams.systolicBP;
sum += this.#coefficients[sex].systolicBP * sbp;

// Systolic blood pressure variability (std dev of repeated measures)
if (normalizedParams.systolicBP_sd) {
 sum += this.#coefficients[sex].systolicBPVariability * normalizedParams.systoli
}

// Total cholesterol to HDL ratio
const cholRatio = normalizedParams.cholesterolRatio ||
 (normalizedParams.totalCholesterol / normalizedParams.hdl);
sum += this.#coefficients[sex].cholesterolRatio * cholRatio;

```

```

// Smoking status
const smokingStatus = normalizedParams.smokingStatus || 'non';
sum += this.#coefficients[sex].smoking[smokingStatus];

// Diabetes status
const diabetesStatus = normalizedParams.diabetesStatus || 'none';
sum += this.#coefficients[sex].diabetes[diabetesStatus];

// Additional risk factors

// CKD (stage 3-5)
if (normalizedParams.chronicKidneyDisease) {
 sum += this.#coefficients[sex].chronicKidneyDisease;
}

// Atrial fibrillation
if (normalizedParams.atrialFibrillation) {
 sum += this.#coefficients[sex].atrialFibrillation;
}

// Migraines
if (normalizedParams.migraines) {
 sum += this.#coefficients[sex].migraines;
}

// Rheumatoid arthritis
if (normalizedParams.rheumatoidArthritis) {
 sum += this.#coefficients[sex].rheumatoidArthritis;
}

// Systemic Lupus erythematosus
if (normalizedParams.systemicLupusErythematosus) {
 sum += this.#coefficients[sex].systemicLupusErythematosus;
}

// Severe mental illness
if (normalizedParams.severeMentalIllness) {
 sum += this.#coefficients[sex].severeMentalIllness;
}

// On blood pressure treatment
if (normalizedParams.onBPMeds) {
 sum += this.#coefficients[sex].bpTreatment;
}

```

```

}

// Erectile dysfunction (men only)
if (sex === 'male' && normalizedParams.erectileDysfunction) {
 sum += this.#coefficients[sex].erectileDysfunction;
}

// Interaction terms

// Age and systolic BP interaction
sum += this.#coefficients[sex].age_systolicBP * age * sbp;

// Age and BMI interaction
if (bmi > 0) {
 sum += this.#coefficients[sex].age_bmi * age * bmi;
}

// Age and BP treatment interaction
if (normalizedParams.onBPMeds) {
 sum += this.#coefficients[sex].age_bpTreatment * age;
}

// Age and diabetes interaction
sum += this.#coefficients[sex].age_diabetes[diabetesStatus] * age;

// Age and smoking interaction
sum += this.#coefficients[sex].age_smoking[smokingStatus] * age;

// 2. Calculate 10-year risk
const s0 = this.#coefficients.survival[sex];
const tenYearRisk = 1 - Math.pow(s0, Math.exp(sum));
const tenYearRiskPercent = Math.round(tenYearRisk * 1000) / 10;

// 3. Calculate heart age and relative risk
const heartAge = this.#calculateHeartAge(normalizedParams, tenYearRisk);
const relativeRisk = this.#calculateRelativeRisk(normalizedParams, tenYearRisk);

// 4. Determine risk category
const riskCategory = this.#getRiskCategory(tenYearRiskPercent);

return {
 success: true,
 tenYearRiskPercent: tenYearRiskPercent,
 riskCategory: riskCategory,
}

```

```

 heartAge: heartAge,
 relativeRisk: relativeRisk,
 algorithm: 'qrisk3',
 parameters: normalizedParams,
 calculationDate: new Date()

);
} catch (error) {
 console.error('Error calculating QRISK3:', error);

 return {
 success: false,
 error: `Calculation error: ${error.message}`,
 algorithm: 'qrisk3'
 };
}

/**
 * Validate required parameters
 * @param {Object} params - Risk parameters
 * @returns {boolean} - Whether all required parameters are present and valid
 * @private
 */
static #validateParameters(params) {
 if (!params) return false;

 const requiredParams = [
 'sex',
 'age',
 'ethnicity',
 'systolicBP'
];

 // Either BMI or both height and weight are required
 if (!params.bmi && (!params.height || !params.weight)) {
 return false;
 }

 // Either cholesterolRatio or both totalCholesterol and hdl are required
 if (!params.cholesterolRatio && (!params.totalCholesterol || !params.hdl)) {
 return false;
 }

 return requiredParams.every(param => {

```

```

 const value = params[param];
 return value !== undefined && value !== null && value !== '';
 });
}

/**
 * Get list of missing parameters
 * @param {Object} params - Risk parameters
 * @returns {Array} - List of missing parameters
 * @private
 */
static #getMissingParameters(params) {
 if (!params) return ['All parameters are missing'];

 const missingParams = [];

 const requiredParams = [
 'sex',
 'age',
 'ethnicity',
 'systolicBP'
];

 requiredParams.forEach(param => {
 const value = params[param];
 if (value === undefined || value === null || value === '') {
 missingParams.push(param);
 }
 });

 // Check BMI or height/weight
 if (!params.bmi && (!params.height || !params.weight)) {
 missingParams.push('BMI or both height and weight');
 }

 // Check cholesterol ratio or total/hdl
 if (!params.cholesterolRatio && (!params.totalCholesterol || !params.hdl)) {
 missingParams.push('Cholesterol ratio or both total cholesterol and HDL');
 }

 return missingParams;
}

```

```

* Normalize parameters
* @param {Object} params - Risk parameters
* @returns {Object} - Normalized parameters
* @private
*/
static #normalizeParameters(params) {
 const normalizedParams = { ...params };

 // Normalize sex
 if (normalizedParams.sex === 'male' || normalizedParams.sex === 'm') {
 normalizedParams.sex = 'male';
 } else if (normalizedParams.sex === 'female' || normalizedParams.sex === 'f') {
 normalizedParams.sex = 'female';
 }

 // Convert string boolean values to actual booleans
 [
 'onBPMeds',
 'familyHistory',
 'chronicKidneyDisease',
 'atrialFibrillation',
 'migraines',
 'rheumatoidArthritis',
 'systemicLupusErythematosus',
 'severeMentalIllness',
 'erectileDysfunction'
].forEach(param => {
 if (normalizedParams[param] === 'true') normalizedParams[param] = true;
 if (normalizedParams[param] === 'false') normalizedParams[param] = false;
 if (normalizedParams[param] === undefined) normalizedParams[param] = false;
 });

 // Convert numeric strings to numbers
 [
 'age',
 'height',
 'weight',
 'bmi',
 'systolicBP',
 'systolicBP_sd',
 'totalCholesterol',
 'hdl',
 'cholesterolRatio'
].forEach(param => {

```

```

 if (normalizedParams[param] !== undefined && normalizedParams[param] !== null) {
 normalizedParams[param] = parseFloat(normalizedParams[param]);
 }
 });

 // Calculate BMI if not provided
 if (!normalizedParams.bmi && normalizedParams.height && normalizedParams.weight) {
 const heightM = normalizedParams.heightUnit === 'in'
 ? normalizedParams.height * 0.0254
 : normalizedParams.height / 100;

 const weightKg = normalizedParams.weightUnit === 'lb'
 ? normalizedParams.weight * 0.453592
 : normalizedParams.weight;

 normalizedParams.bmi = Math.round((weightKg / (heightM * heightM)) * 10) / 10;
 }

 // Calculate cholesterol ratio if not provided
 if (!normalizedParams.cholesterolRatio && normalizedParams.totalCholesterol && normalizedParams.hdl) {
 normalizedParams.cholesterolRatio =
 Math.round((normalizedParams.totalCholesterol / normalizedParams.hdl) * 100) / 100;
 }

 // Default ethnicity if not specified
 if (!normalizedParams.ethnicity) {
 normalizedParams.ethnicity = 'white';
 }

 // Default smoking status if not specified
 if (!normalizedParams.smokingStatus) {
 normalizedParams.smokingStatus = normalizedParams.isSmoker ? 'light' : 'non';
 }

 // Default diabetes status if not specified
 if (!normalizedParams.diabetesStatus) {
 normalizedParams.diabetesStatus = normalizedParams.hasDiabetes ? 'type2' : 'none';
 }

 return normalizedParams;
}

/**
 * Calculate heart age based on risk

```

```

* @param {Object} params - Normalized parameters
* @param {number} risk - 10-year risk
* @returns {number} - Heart age in years
* @private
*/
static #calculateHeartAge(params, risk) {
 const sex = params.sex;
 let heartAge = params.age;

 // Create a reference individual with same risk factors but optimal modifiable factors
 const reference = { ...params };

 // Set optimal modifiable risk factors
 reference.systolicBP = 120;
 reference.systolicBP_sd = 0;
 reference.cholesterolRatio = 3.5;
 reference.smokingStatus = 'non';
 reference.onBPMeds = false;

 // Find age that gives equivalent risk
 const tolerance = 0.001;
 const maxIterations = 100;
 let iterations = 0;
 let currentRisk = risk;
 let targetRisk = risk;
 let delta = 1;

 // Binary search to find heart age
 while (Math.abs(delta) > tolerance && iterations < maxIterations) {
 iterations++;

 // Set reference age
 reference.age = heartAge;

 // Calculate risk for reference
 const refParams = this.#normalizeParameters(reference);

 // Calculate sum for reference using same algorithm as main calculation
 let sum = 0;

 // Apply age terms
 sum += this.#coefficients[sex].age1 * refParams.age;
 sum += this.#coefficients[sex].age2 * refParams.age * refParams.age;
 }
}

```

```

// BMI terms
if (refParams.bmi > 0) {
 sum += this.#coefficients[sex].bmi1 * refParams.bmi;
 sum += this.#coefficients[sex].bmi2 * refParams.bmi * refParams.bmi;
}

// Apply other factors
sum += this.#coefficients[sex].ethnicOrigin[refParams.ethnicity];

if (refParams.familyHistory) {
 sum += this.#coefficients[sex].familyHistoryCHD;
}

sum += this.#coefficients[sex].systolicBP * refParams.systolicBP;

if (refParams.systolicBP_sd) {
 sum += this.#coefficients[sex].systolicBPVariability * refParams.systolicBP_sd;
}

sum += this.#coefficients[sex].cholesterolRatio * refParams.cholesterolRatio;
sum += this.#coefficients[sex].smoking[refParams.smokingStatus];
sum += this.#coefficients[sex].diabetes[refParams.diabetesStatus];

// Add additional risk factors
if (refParams.chronicKidneyDisease) {
 sum += this.#coefficients[sex].chronicKidneyDisease;
}

if (refParams.atrialFibrillation) {
 sum += this.#coefficients[sex].atrialFibrillation;
}

if (refParams.migraines) {
 sum += this.#coefficients[sex].migraines;
}

if (refParams.rheumatoidArthritis) {
 sum += this.#coefficients[sex].rheumatoidArthritis;
}

if (refParams.systemicLupusErythematosus) {
 sum += this.#coefficients[sex].systemicLupusErythematosus;
}

```

```

if (refParams.severeMentalIllness) {
 sum += this.#coefficients[sex].severeMentalIllness;
}

if (refParams.onBPMeds) {
 sum += this.#coefficients[sex].bpTreatment;
}

if (sex === 'male' && refParams.erectileDysfunction) {
 sum += this.#coefficients[sex].erectileDysfunction;
}

// Interaction terms
sum += this.#coefficients[sex].age_systolicBP * refParams.age * refParams.systolicBP;

if (refParams.bmi > 0) {
 sum += this.#coefficients[sex].age_bmi * refParams.age * refParams.bmi;
}

if (refParams.onBPMeds) {
 sum += this.#coefficients[sex].age_bpTreatment * refParams.age;
}

sum += this.#coefficients[sex].age_diabetes[refParams.diabetesStatus] * refParams.age;
sum += this.#coefficients[sex].age_smoking[refParams.smokingStatus] * refParams.age;

// Calculate reference risk
const s0 = this.#coefficients.survival[sex];
const refRisk = 1 - Math.pow(s0, Math.exp(sum));

// Adjust heart age
delta = refRisk - targetRisk;

if (delta > 0) {
 heartAge -= Math.max(0.1, Math.abs(delta) * 10);
} else {
 heartAge += Math.max(0.1, Math.abs(delta) * 10);
}

currentRisk = refRisk;
}

// Round heart age to nearest year
return Math.round(heartAge);

```

```

}

/**
 * Calculate relative risk compared to a person of same age/sex with optimal risk factors
 * @param {Object} params - Normalized parameters
 * @param {number} risk - 10-year risk
 * @returns {number} - Relative risk ratio
 * @private
 */
static #calculateRelativeRisk(params, risk) {
 const sex = params.sex;

 // Create a reference individual with same age/sex but optimal modifiable factors
 const reference = { ...params };

 // Set optimal modifiable risk factors
 reference.systolicBP = 120;
 reference.systolicBP_sd = 0;
 reference.cholesterolRatio = 3.5;
 reference.smokingStatus = 'non';
 reference.onBPMeds = false;
 reference.bmi = 22.5;

 // Calculate risk for reference
 const refParams = this.#normalizeParameters(reference);

 // Calculate sum for reference using same algorithm as main calculation
 let sum = 0;

 // Apply age terms
 sum += this.#coefficients[sex].age1 * refParams.age;
 sum += this.#coefficients[sex].age2 * refParams.age * refParams.age;

 // BMI terms
 if (refParams.bmi > 0) {
 sum += this.#coefficients[sex].bmi1 * refParams.bmi;
 sum += this.#coefficients[sex].bmi2 * refParams.bmi * refParams.bmi;
 }

 // Apply other factors
 sum += this.#coefficients[sex].ethnicOrigin[refParams.ethnicity];

 if (refParams.familyHistory) {
 sum += this.#coefficients[sex].familyHistoryCHD;
 }
}

```

```

}

sum += this.#coefficients[sex].systolicBP * refParams.systolicBP;

if (refParams.systolicBP_sd) {
 sum += this.#coefficients[sex].systolicBPVariability * refParams.systolicBP_sd;
}

sum += this.#coefficients[sex].cholesterolRatio * refParams.cholesterolRatio;
sum += this.#coefficients[sex].smoking[refParams.smokingStatus];
sum += this.#coefficients[sex].diabetes[refParams.diabetesStatus];

// Add additional risk factors
if (refParams.chronicKidneyDisease) {
 sum += this.#coefficients[sex].chronicKidneyDisease;
}

if (refParams.atrialFibrillation) {
 sum += this.#coefficients[sex].atrialFibrillation;
}

if (refParams.migraines) {
 sum += this.#coefficients[sex].migraines;
}

if (refParams.rheumatoidArthritis) {
 sum += this.#coefficients[sex].rheumatoidArthritis;
}

if (refParams.systemicLupusErythematosus) {
 sum += this.#coefficients[sex].systemicLupusErythematosus;
}

if (refParams.severeMentalIllness) {
 sum += this.#coefficients[sex].severeMentalIllness;
}

if (refParams.onBPMeds) {
 sum += this.#coefficients[sex].bpTreatment;
}

if (sex === 'male' && refParams.erectileDysfunction) {
 sum += this.#coefficients[sex].erectileDysfunction;
}

```

```

// Interaction terms
sum += this.#coefficients[sex].age_systolicBP * refParams.age * refParams.systolicBP;

if (refParams.bmi > 0) {
 sum += this.#coefficients[sex].age_bmi * refParams.age * refParams.bmi;
}

if (refParams.onBPMeds) {
 sum += this.#coefficients[sex].age_bpTreatment * refParams.age;
}

sum += this.#coefficients[sex].age_diabetes[refParams.diabetesStatus] * refParams.age;
sum += this.#coefficients[sex].age_smoking[refParams.smokingStatus] * refParams.age;

// Calculate reference risk
const s0 = this.#coefficients.survival[sex];
const referenceRisk = 1 - Math.pow(s0, Math.exp(sum));

// Calculate relative risk (if reference risk is 0, avoid division by zero)
const relativeRisk = referenceRisk > 0 ? risk / referenceRisk : 1;

// Round to 2 decimal places
return Math.round(relativeRisk * 100) / 100;
}

/**
 * Get risk category based on percentage
 * @param {number} riskPercent - Risk percentage
 * @returns {string} - Risk category
 * @private
 */
static #getRiskCategory(riskPercent) {
 if (riskPercent < this.#riskCategories.low.max) {
 return 'low';
 } else if (riskPercent < this.#riskCategories.intermediate.max) {
 return 'intermediate';
 } else {
 return 'high';
 }
}

export default QRISK3Calculator;

```

```

```
    if (this.#initialized) return true;

try {
    console.log('Initializing secure storage...');

    // Check if localStorage and crypto are available
    if (!window.localStorage || !window.crypto) {
        console.error('Secure storage requires localStorage and crypto APIs');
        return false;
    }

    // Try to retrieve an existing key from session storage
    const storedKey = sessionStorage.getItem(`#${this.#storagePrefix}encryption-key`);

    if (storedKey) {
        // Convert stored key from base64 to Uint8Array
        this.#encryptionKey = this.#base64ToArrayBuffer(storedKey);
    } else {
        // Generate a new encryption key
        this.#encryptionKey = await this.#generateEncryptionKey();

        // Store key in session storage (will be lost when the session ends)
        sessionStorage.setItem(
            `#${this.#storagePrefix}encryption-key`,
            this.#arrayBufferToBase64(this.#encryptionKey)
        );
    }
}

this.#initialized = true;
return true;
} catch (error) {
    console.error('Error initializing secure storage:', error);
    return false;
}
}

/**
 * Store encrypted data in localStorage
 * @param {string} key - Storage key
 * @param {*} data - Data to store (will be stringified)
 * @returns {Promise<boolean>} - Whether storage was successful
 */
static asyncsetItem(key, data) {
    if (!this.#initialized) {

```

```
    await this.initialize();
}

if (!this.#encryptionKey) {
    console.error('Encryption key not available');
    return false;
}

try {
    // Prepare data
    const dataString = typeof data === 'string' ? data : JSON.stringify(data);
    const dataBuffer = new TextEncoder().encode(dataString);

    // Generate initialization vector
    const iv = crypto.getRandomValues(new Uint8Array(12));

    // Encrypt data
    const encryptedData = await window.crypto.subtle.encrypt(
        {
            name: 'AES-GCM',
            iv: iv
        },
        this.#encryptionKey,
        dataBuffer
    );

    // Prepare storage object
    const storageObject = {
        iv: this.#arrayBufferToBase64(iv),
        data: this.#arrayBufferToBase64(encryptedData),
        timestamp: Date.now()
    };

    // Store the encrypted data
    localStorage.setItem(
        `${this.#storagePrefix}${key}`,
        JSON.stringify(storageObject)
    );
}

return true;
} catch (error) {
    console.error(`Error storing data for key "${key}"`, error);
    return false;
}
```

```
}

/**
 * Retrieve and decrypt data from localStorage
 * @param {string} key - Storage key
 * @param {*} defaultValue - Default value if key doesn't exist
 * @returns {Promise<*>} - Decrypted data or defaultValue
 */
static async getItem(key, defaultValue = null) {
    if (!this.#initialized) {
        await this.initialize();
    }

    if (!this.#encryptionKey) {
        console.error('Encryption key not available');
        return defaultValue;
    }

    try {
        // Get encrypted data from localStorage
        const storedData = localStorage.getItem(`#${this.#storagePrefix}${key}`);

        if (!storedData) {
            return defaultValue;
        }

        // Parse storage object
        const storageObject = JSON.parse(storedData);

        // Convert from base64
        const iv = this.#base64ToArrayBuffer(storageObject.iv);
        const encryptedData = this.#base64ToArrayBuffer(storageObject.data);

        // Decrypt data
        const decryptedBuffer = await window.crypto.subtle.decrypt(
            {
                name: 'AES-GCM',
                iv: iv
            },
            this.#encryptionKey,
            encryptedData
        );

        // Convert to string
    }
}
```

```

        const decryptedString = new TextDecoder().decode(decryptedBuffer);

        // Try to parse as JSON
        try {
            return JSON.parse(decryptedString);
        } catch (e) {
            // Return as string if not valid JSON
            return decryptedString;
        }
    } catch (error) {
        console.error(`Error retrieving data for key "${key}":`, error);
        return defaultValue;
    }
}

/***
 * Remove data from localStorage
 * @param {string} key - Storage key
 * @returns {boolean} - Whether removal was successful
 */
static removeItem(key) {
    try {
        localStorage.removeItem(`${this.#storagePrefix}${key}`);
        return true;
    } catch (error) {
        console.error(`Error removing data for key "${key}":`, error);
        return false;
    }
}

/***
 * Clear all secure storage items
 * @returns {boolean} - Whether clearing was successful
 */
static clear() {
    try {
        // Get all keys starting with the storage prefix
        const keys = [];
        for (let i = 0; i < localStorage.length; i++) {
            const key = localStorage.key(i);
            if (key.startsWith(this.#storagePrefix)) {
                keys.push(key);
            }
        }
    }
}

```

```

    // Remove all matching keys
    keys.forEach(key => localStorage.removeItem(key));

    return true;
} catch (error) {
    console.error('Error clearing secure storage:', error);
    return false;
}
}

/**
 * Generate a new AES-GCM encryption key
 * @returns {Promise<CryptoKey>} - Generated key
 * @private
 */
static async #generateEncryptionKey() {
    return window.crypto.subtle.generateKey(
        {
            name: 'AES-GCM',
            length: 256
        },
        true,
        ['encrypt', 'decrypt']
    );
}

/**
 * Convert ArrayBuffer to Base64 string
 * @param {ArrayBuffer} buffer - ArrayBuffer to convert
 * @returns {string} - Base64 string
 * @private
 */
static #arrayBufferToBase64(buffer) {
    const bytes = new Uint8Array(buffer);
    let binary = '';

    for (let i = 0; i < bytes.byteLength; i++) {
        binary += String.fromCharCode(bytes[i]);
    }

    return window.btoa(binary);
}

```

```
/**  
 * Convert Base64 string to ArrayBuffer  
 * @param {string} base64 - Base64 string to convert  
 * @returns {ArrayBuffer} - ArrayBuffer  
 * @private  
 */  
  
static #base64ToArrayBuffer(base64) {  
    const binaryString = window.atob(base64);  
    const bytes = new Uint8Array(binaryString.length);  
  
    for (let i = 0; i < binaryString.length; i++) {  
        bytes[i] = binaryString.charCodeAt(i);  
    }  
  
    return bytes.buffer;  
}  
}  
  
export default SecureStorage;
```

1.12 xss-protection.js

javascript

```

/**
 * XSS Protection for CVD Risk Toolkit
 *
 * Provides input sanitization and protection against
 * cross-site scripting (XSS) attacks.
 */

class XSSProtection {
    /**
     * Sanitize a string to prevent XSS
     * @param {string} input - Input string
     * @returns {string} - Sanitized string
     */
    static sanitize(input) {
        if (!input) return '';

        if (typeof input !== 'string') {
            input = String(input);
        }

        // Replace HTML special characters with entities
        return input
            .replace(/&/g, '&')
            .replace(/</g, '<')
            .replace(/>/g, '>')
            .replace(/"/g, '"')
            .replace(/\'/g, ''')
            .replace(/\//g, '/');
    }

    /**
     * Sanitize an object's string properties recursively
     * @param {Object} obj - Object to sanitize
     * @returns {Object} - Sanitized object
     */
    static sanitizeObject(obj) {
        if (!obj || typeof obj !== 'object') return obj;

        // Handle arrays
        if (Array.isArray(obj)) {
            return obj.map(item => this.sanitizeObject(item));
        }
    }
}

```

```

// Process object properties
const result = {};

for (const [key, value] of Object.entries(obj)) {
    if (typeof value === 'string') {
        result[key] = this.sanitize(value);
    } else if (typeof value === 'object' && value !== null) {
        result[key] = this.sanitizeObject(value);
    } else {
        result[key] = value;
    }
}

return result;
}

/**
 * Sanitize form inputs to prevent XSS
 * @param {HTMLFormElement} form - Form element
 * @returns {Object} - Sanitized form data
 */
static sanitizeForm(form) {
    if (!form || !(form instanceof HTMLFormElement)) {
        console.error('Invalid form element');
        return {};
    }

    const formData = new FormData(form);
    const sanitizedData = {};

    for (const [key, value] of formData.entries()) {
        if (typeof value === 'string') {
            sanitizedData[key] = this.sanitize(value);
        } else {
            sanitizedData[key] = value;
        }
    }

    // Add checkbox values not included in FormData when unchecked
    const checkboxes = form.querySelectorAll('input[type="checkbox"]');
    checkboxes.forEach(checkbox => {
        if (!formData.has(checkbox.name)) {
            sanitizedData[checkbox.name] = false;
        } else {

```

```

        sanitizedData[checkbox.name] = true;
    }
});

return sanitizedData;
}

/** 
 * Safely add HTML content to an element
 * @param {HTMLElement} element - Target element
 * @param {string} html - HTML content
 * @param {boolean} sanitize - Whether to sanitize the HTML
 */
static setElementHTML(element, html, sanitize = true) {
    if (!element || !(element instanceof HTMLElement)) {
        console.error('Invalid target element');
        return;
    }

    if (sanitize) {
        element.textContent = html; // This automatically sanitizes
    } else {
        element.innerHTML = html;
    }
}

/** 
 * Create a DOM element with sanitized attributes
 * @param {string} tag - Element tag
 * @param {Object} attributes - Element attributes
 * @param {string|Node} content - Element content
 * @returns {HTMLElement} - Created element
 */
static createElement(tag, attributes = {}, content = '') {
    const element = document.createElement(tag);

    // Add sanitized attributes
    for (const [key, value] of Object.entries(attributes)) {
        if (key.toLowerCase().startsWith('on')) {
            console.warn(`Blocked potentially unsafe attribute: ${key}`);
            continue;
        }

        element.setAttribute(key, this.sanitize(value));
    }
}

```

```

}

// Add content
if (content instanceof Node) {
    element.appendChild(content);
} else if (typeof content === 'string') {
    element.textContent = content;
}

return element;
}

/**
 * Set up form protection
 * @param {HTMLFormElement} form - Form to protect
 */
static protectForm(form) {
    if (!form || !(form instanceof HTMLFormElement)) {
        console.error('Invalid form element');
        return;
    }

    // Generate a random token
    const csrfToken = this.#generateRandomToken();

    // Add hidden token field to the form
    const tokenField = document.createElement('input');
    tokenField.type = 'hidden';
    tokenField.name = 'csrf_token';
    tokenField.value = csrfToken;
    form.appendChild(tokenField);

    // Store token in session storage
    sessionStorage.setItem(`csrf_token_${form.id || form.name}`, csrfToken);

    // Add submission validation
    form.addEventListener('submit', (event) => {
        const submittedToken = tokenField.value;
        const storedToken = sessionStorage.getItem(`csrf_token_${form.id || form.name}`);

        if (!submittedToken || submittedToken !== storedToken) {
            event.preventDefault();
            console.error('CSRF token validation failed');
        }
    });
}

```

```

    // Reset token
    const newToken = this.#generateRandomToken();
    tokenField.value = newToken;
    sessionStorage.setItem(`csrf_token_${form.id || form.name}`, newToken);

    return false;
}

// Sanitize form data before submission
this.sanitizeForm(form);
});

}

/**
 * Generate a random token
 * @returns {string} - Random token
 * @private
 */
static #generateRandomToken() {
    const array = new Uint8Array(16);
    window.crypto.getRandomValues(array);
    return Array.from(array, byte => byte.toString(16).padStart(2, '0')).join('');
}

export default XSSProtection;

```

2. Calculation Algorithms

2.1 framingham-algorithm.js

javascript

```

/**
 * Framingham Risk Score Calculator for CVD Risk Toolkit
 *
 * Implements the 2008 Framingham Heart Study General
 * Cardiovascular Disease Risk Score algorithm.
 *
 * References:
 * D'Agostino RB Sr, Vasan RS, Pencina MJ, et al.
 * General cardiovascular risk profile for use in primary care:
 * the Framingham Heart Study. Circulation. 2008;117(6):743-753.
 */

class FraminghamRiskScore {
    // Coefficient tables from the 2008 Framingham paper
    static #coefficients = {
        // Beta coefficients for women
        female: {
            ln_age: 2.32888,
            ln_age_squared: 0,
            ln_total_cholesterol: 1.20904,
            ln_hdl_cholesterol: -0.70833,
            ln_sbp_not_treated: 2.76157,
            ln_sbp_treated: 2.82263,
            smoker: 0.52873,
            diabetes: 0.69154
        },
        // Beta coefficients for men
        male: {
            ln_age: 3.06117,
            ln_age_squared: 0,
            ln_total_cholesterol: 1.12370,
            ln_hdl_cholesterol: -0.93263,
            ln_sbp_not_treated: 1.93303,
            ln_sbp_treated: 1.99881,
            smoker: 0.65451,
            diabetes: 0.57367
        },
        // Survival function values at 10 years
        survival: {
            female: 0.95012,
            male: 0.88936
        },
        // Mean values of risk factors from the Framingham cohort
    }
}

```

```

means: {
    female: {
        ln_age: 3.8686,
        ln_total_cholesterol: 5.2146,
        ln_hdl_cholesterol: 4.0132,
        ln_sbp_not_treated: 4.7958,
        ln_sbp_treated: 4.7958,
        smoker: 0.3460,
        diabetes: 0.0437
    },
    male: {
        ln_age: 3.8371,
        ln_total_cholesterol: 5.3223,
        ln_hdl_cholesterol: 3.7731,
        ln_sbp_not_treated: 4.7835,
        ln_sbp_treated: 4.7835,
        smoker: 0.3379,
        diabetes: 0.0431
    }
},
};

// Risk categories based on Canadian guidelines
static #riskCategories = {
    low: { max: 10 },
    intermediate: { min: 10, max: 20 },
    high: { min: 20 }
};

/***
 * Calculate Framingham Risk Score
 * @param {Object} params - Risk parameters
 * @returns {Object} - Risk calculation results
 */
static calculateRisk(params) {
    try {
        console.log('Calculating Framingham Risk Score with parameters:', params);

        // Validate required parameters
        if (!this.#validateParameters(params)) {
            return {
                success: false,
                error: 'Missing or invalid required parameters',
                missingParams: this.#getMissingParameters(params),
            };
        }

        const riskScore = calculateFraminghamRiskScore(params);
        const riskCategory = determineRiskCategory(riskScore);
        const riskProfile = generateRiskProfile(riskScore, riskCategory);
        const riskSummary = generateRiskSummary(riskScore, riskCategory);
        const riskConclusion = generateRiskConclusion(riskScore, riskCategory);

        return {
            success: true,
            riskScore: riskScore,
            riskCategory: riskCategory,
            riskProfile: riskProfile,
            riskSummary: riskSummary,
            riskConclusion: riskConclusion,
        };
    } catch (error) {
        return {
            success: false,
            error: 'An error occurred while calculating the risk score: ' + error.message,
        };
    }
}

```

```

        algorithm: 'framingham'
    };

}

// Normalize parameters
const normalizedParams = this.#normalizeParameters(params);
const sex = normalizedParams.sex;

// Calculate individual components
let sum = 0;

// Age component
sum += this.#coefficients[sex].ln_age * Math.log(normalizedParams.age);

// Total cholesterol component
sum += this.#coefficients[sex].ln_total_cholesterol *
    Math.log(normalizedParams.totalCholesterol);

// HDL component
sum += this.#coefficients[sex].ln_hdl_cholesterol *
    Math.log(normalizedParams.hdl);

// Systolic BP component, based on treatment status
if (normalizedParams.onBPMeds) {
    sum += this.#coefficients[sex].ln_sbp_treated *
        Math.log(normalizedParams.systolicBP);
} else {
    sum += this.#coefficients[sex].ln_sbp_not_treated *
        Math.log(normalizedParams.systolicBP);
}

// Smoking component
if (normalizedParams.isSmoker) {
    sum += this.#coefficients[sex].smoker;
}

// Diabetes component
if (normalizedParams.hasDiabetes) {
    sum += this.#coefficients[sex].diabetes;
}

// Calculate mean sum for the reference population
let meanSum = 0;

```

```

meanSum += this.#coefficients[sex].ln_age *
    this.#coefficients.means[sex].ln_age;

meanSum += this.#coefficients[sex].ln_total_cholesterol *
    this.#coefficients.means[sex].ln_total_cholesterol;

meanSum += this.#coefficients[sex].ln_hdl_cholesterol *
    this.#coefficients.means[sex].ln_hdl_cholesterol;

if (normalizedParams.onBPMeds) {
    meanSum += this.#coefficients[sex].ln_sbp_treated *
        this.#coefficients.means[sex].ln_sbp_treated;
} else {
    meanSum += this.#coefficients[sex].ln_sbp_not_treated *
        this.#coefficients.means[sex].ln_sbp_not_treated;
}

if (this.#coefficients.means[sex].smoker > 0) {
    meanSum += this.#coefficients[sex].smoker *
        this.#coefficients.means[sex].smoker;
}

if (this.#coefficients.means[sex].diabetes > 0) {
    meanSum += this.#coefficients[sex].diabetes *
        this.#coefficients.means[sex].diabetes;
}

// Calculate 10-year risk
const s0 = this.#coefficients.survival[sex];
const riskFactor = Math.exp(sum - meanSum);
const tenYearRisk = 1 - Math.pow(s0, riskFactor);
const tenYearRiskPercent = Math.round(tenYearRisk * 1000) / 10;

// Apply risk modifiers if present
let modifiedRiskPercent = tenYearRiskPercent;
const riskModifiers = [];

// Family history modifier (increases risk by 1.5x)
if (normalizedParams.familyHistory) {
    modifiedRiskPercent *= 1.5;
    riskModifiers.push({
        factor: 'Family history of premature CVD',
        multiplier: 1.5,
        source: 'CCS 2021 Guidelines'
    });
}

```

```

    });
}

// South Asian ethnicity modifier (increases risk by 1.5x)
if (normalizedParams.isSouthAsian) {
    modifiedRiskPercent *= 1.5;
    riskModifiers.push({
        factor: 'South Asian ancestry',
        multiplier: 1.5,
        source: 'CCS 2021 Guidelines'
    });
}

// Lipoprotein(a) modifier (increases risk if > 50 mg/dL or > 100 nmol/L)
if (normalizedParams.lpa) {
    const lpaThreshold = normalizedParams.lpaUnit === 'nmol/L' ? 100 : 50;

    if (normalizedParams.lpa > lpaThreshold) {
        modifiedRiskPercent *= 1.3;
        riskModifiers.push({
            factor: 'Elevated Lipoprotein(a)',
            multiplier: 1.3,
            source: 'CCS 2021 Guidelines'
        });
    }
}

// Round modified risk
modifiedRiskPercent = Math.round(modifiedRiskPercent * 10) / 10;

// Determine risk category
const riskCategory = this.#getRiskCategory(modifiedRiskPercent);

return {
    success: true,
    tenYearRiskPercent: tenYearRiskPercent,
    modifiedRiskPercent: modifiedRiskPercent,
    riskCategory: riskCategory,
    riskModifiers: riskModifiers,
    algorithm: 'framingham',
    parameters: normalizedParams,
    calculationDate: new Date()
};
} catch (error) {
}

```

```

        console.error('Error calculating Framingham risk:', error);

    return {
        success: false,
        error: `Calculation error: ${error.message}`,
        algorithm: 'framingham'
    };
}

/**
 * Validate required parameters
 * @param {Object} params - Risk parameters
 * @returns {boolean} - Whether all required parameters are present and valid
 * @private
 */
static #validateParameters(params) {
    if (!params) return false;

    const requiredParams = [
        'sex',
        'age',
        'totalCholesterol',
        'hdl',
        'systolicBP'
    ];

    return requiredParams.every(param => {
        const value = params[param];
        return value !== undefined && value !== null && value !== '';
    });
}

/**
 * Get list of missing parameters
 * @param {Object} params - Risk parameters
 * @returns {Array} - List of missing parameters
 * @private
 */
static #getMissingParameters(params) {
    if (!params) return ['All parameters are missing'];

    const requiredParams = [
        'sex',

```

```

        'age',
        'totalCholesterol',
        'hdl',
        'systolicBP'
    ];

    return requiredParams.filter(param => {
        const value = params[param];
        return value === undefined || value === null || value === '';
    });
}

/**
 * Normalize parameters
 * @param {Object} params - Risk parameters
 * @returns {Object} - Normalized parameters
 * @private
 */
static #normalizeParameters(params) {
    const normalizedParams = { ...params };

    // Normalize sex
    if (normalizedParams.sex === 'male' || normalizedParams.sex === 'm') {
        normalizedParams.sex = 'male';
    } else if (normalizedParams.sex === 'female' || normalizedParams.sex === 'f') {
        normalizedParams.sex = 'female';
    }

    // Convert string boolean values to actual booleans
    ['onBPMeds', 'isSmoker', 'hasDiabetes', 'familyHistory', 'isSouthAsian'].forEach(param => {
        if (normalizedParams[param] === 'true') normalizedParams[param] = true;
        if (normalizedParams[param] === 'false') normalizedParams[param] = false;
        if (normalizedParams[param] === undefined) normalizedParams[param] = false;
    });

    // Convert numeric strings to numbers
    ['age', 'totalCholesterol', 'hdl', 'ldl', 'systolicBP', 'lpa'].forEach(param => {
        if (normalizedParams[param] !== undefined && normalizedParams[param] !== null) {
            normalizedParams[param] = parseFloat(normalizedParams[param]);
        }
    });

    // Default lpa unit if not specified
    if (normalizedParams.lpa && !normalizedParams.lpaUnit) {

```

```

        normalizedParams.lpaUnit = 'mg/dL';
    }

    return normalizedParams;
}

/**
 * Get risk category based on percentage
 * @param {number} riskPercent - Risk percentage
 * @returns {string} - Risk category
 * @private
 */
static #getRiskCategory(riskPercent) {
    if (riskPercent < this.#riskCategories.low.max) {
        return 'low';
    } else if (riskPercent < this.#riskCategories.intermediate.max) {
        return 'intermediate';
    } else {
        return 'high';
    }
}

export default FraminghamRiskScore;
```
Load modules specific to a view
```
 * @param {string} viewName - View name
 * @returns {Promise<Object>} - Object containing loaded modules
 */
static async loadViewModules(viewName) {
    console.log(`Loading modules for view: ${viewName}`);

    const viewModules = {
        // FRS calculator view
        frs: [
            './calculations/framingham-algorithm.js'
        ],

        // QRISK3 calculator view
        qrisk: [
            './calculations/qrisk3-algorithm.js'
        ],

        // Combined view
        combined: [

```

```

        './calculations/framingham-algorithm.js',
        './calculations/qrisk3-algorithm.js',
        './visualizations/combined-view-manager.js',
        './visualizations/chart-renderer.js'
    ],

    // Medication view
    medication: [
        './calculations/treatment-recommendations.js'
    ],

    // Advanced visualization view
    'advanced-viz': [
        './visualizations/chart-renderer.js',
        './visualizations/chart-exporter.js'
    ],

    // Data import/export view
    data: [
        './utils/data-import-export.js',
        './utils/pdf-generator.js'
    ]
};

// Get modules for view
const modules = viewModules[viewName] || [];

if (modules.length === 0) {
    console.warn(`No specific modules defined for view: ${viewName}`);
    return {};
}

// Load modules
try {
    const loadedModules = {};

    await Promise.all(modules.map(async (modulePath) => {
        const moduleName = modulePath.split('/').pop().replace('.js', '');
        loadedModules[moduleName] = await this.loadModule(modulePath);
    }));
}

console.log(`Loaded ${Object.keys(loadedModules).length} modules for view: ${viewName}`);
return loadedModules;
} catch (error) {
}

```

```

        console.error(`Error loading modules for view ${viewName}:`, error);
        ErrorDetectionSystem.showErrorNotification(
            new Error(`Failed to load modules for ${viewName} view. Some features may be unavailable due to module loading error`)
        );
        return {};
    }

}

/***
 * Create a fallback module
 * @param {string} modulePath - Module path
 * @param {Error} error - Loading error
 * @returns {Object} - Fallback module
 */
static #createFallbackModule(modulePath, error) {
    const moduleName = modulePath.split('/').pop().replace('.js', '');

    console.warn(`Creating fallback for module: ${moduleName}`);

    // Create different fallbacks based on module type
    switch (moduleName) {
        case 'framingham-algorithm':
            return {
                calculateRisk: (params) => {
                    console.warn('Using fallback Framingham calculator - results are approximate');
                    return {
                        success: true,
                        tenYearRiskPercent: 7.5,
                        riskCategory: 'intermediate',
                        message: 'Fallback calculation - please recalculate when online',
                        algorithm: 'framingham',
                        calculationDate: new Date()
                    };
                }
            };
        case 'qrisk3-algorithm':
            return {
                calculateRisk: (params) => {
                    console.warn('Using fallback QRISK3 calculator - results are approximate');
                    return {
                        success: true,
                        tenYearRiskPercent: 7.5,
                    };
                }
            };
    }
}

```

```

        riskCategory: 'intermediate',
        message: 'Fallback calculation - please recalculate when online',
        algorithm: 'qrisk3',
        calculationDate: new Date()
    );
}
};

case 'treatment-recommendations':
return {
    generateRecommendations: () => {
        return {
            success: true,
            ldlTarget: '< 2.0 mmol/L',
            primaryTreatment: 'Moderate to high intensity statin',
            alternativeTreatments: 'Ezetimibe, PCSK9 inhibitor',
            message: 'Fallback recommendations - please recalculate when online'
        };
    },
    generateFRSRecommendations: () => ({
        summary: 'Fallback recommendations - please recalculate when online'
    }),
    generateQRISKRecommendations: () => ({
        summary: 'Fallback recommendations - please recalculate when online'
    })
};

case 'chart-renderer':
return {
    renderChart: () => {
        return Promise.resolve({
            success: false,
            message: 'Chart rendering is unavailable in fallback mode',
            details: 'The chart rendering module could not be loaded. Please try again later.'
        });
    }
};

case 'data-import-export':
return {
    importData: () => Promise.reject(new Error('Import functionality unavailable')),
    exportData: () => Promise.reject(new Error('Export functionality unavailable')),
    loadImportedData: () => Promise.reject(new Error('Import functionality unavailable'))
};

```

```

default:
    // Generic fallback
    return {
        __error: error,
        __modulePath: modulePath,
        __isFallback: true
    };
}

/**
 * Clear module cache
 * @param {string} modulePath - Optional module path to clear
 */
static clearCache(modulePath = null) {
    if (modulePath) {
        this.#loadedModules.delete(modulePath);
    } else {
        this.#loadedModules.clear();
    }
}

export default ModuleLoader;

```

1.10 batch-processor.js

javascript

```
/**  
 * Batch Processor for CVD Risk Toolkit  
 *  
 * Manages batch processing for data operations and  
 * concurrent task execution with rate limiting.  
 */  
  
class BatchProcessor {  
    /**  
     * Process items in batches  
     * @param {Array} items - Items to process  
     * @param {Function} processFn - Processing function  
     * @param {Object} options - Options  
     * @returns {Promise<Object>} - Processing results  
     */  
    static async processBatch(items, processFn, {  
        batchSize = 10,  
        delay = 0,  
        concurrency = 1,  
        onProgress = null  
    } = {}) {  
        if (!items || items.length === 0) {  
            return { success: true, count: 0, results: [] };  
        }  
  
        console.log(`Processing ${items.length} items in batches of ${batchSize}`);  
  
        const results = [];  
        let successCount = 0;  
        let failureCount = 0;  
        let processedCount = 0;  
  
        // Process in batches  
        for (let i = 0; i < items.length; i += batchSize) {  
            const batch = items.slice(i, i + batchSize);  
  
            // Process batch with concurrency limit  
            const batchResults = await this.#processConcurrent(  
                batch,  
                processFn,  
                concurrency  
            );
```

```

// Process results
for (let j = 0; j < batchResults.length; j++) {
  const result = batchResults[j];
  const item = batch[j];

  processedCount++;

  if (result.status === 'fulfilled') {
    successCount++;
    results.push({
      success: true,
      item,
      result: result.value
    });
  } else {
    failureCount++;
    results.push({
      success: false,
      item,
      error: result.reason
    });
  }

  console.error('Failed to process item:', item, result.reason);
}

// Call progress callback if provided
if (onProgress) {
  onProgress({
    processedCount,
    totalCount: items.length,
    successCount,
    failureCount,
    percentComplete: Math.round((processedCount / items.length) * 100)
  });
}

// Add delay between batches if specified
if (delay > 0 && i + batchSize < items.length) {
  await new Promise(resolve => setTimeout(resolve, delay));
}

return {

```

```

        success: failureCount === 0,
        count: processedCount,
        successCount,
        failureCount,
        results
    );
}

/** 
 * Process items concurrently with limited concurrency
 * @param {Array} items - Items to process
 * @param {Function} processFn - Processing function
 * @param {number} concurrency - Maximum concurrency
 * @returns {Promise<Array>} - Processing results
 * @private
 */
static async #processConcurrent(items, processFn, concurrency) {
    // Process all items at once if concurrency is unlimited
    if (concurrency <= 0 || concurrency >= items.length) {
        return Promise.allSettled(items.map(item => processFn(item)));
    }

    const results = [];
    let index = 0;

    // Initial batch of promises
    const initialPromises = items.slice(0, concurrency).map((item, i) => {
        return processFn(item)
            .then(result => {
                results[i] = { status: 'fulfilled', value: result };
                return i;
            })
            .catch(error => {
                results[i] = { status: 'rejected', reason: error };
                return i;
            });
    });

    // Queue of active promises
    const queue = [...initialPromises];
    index = concurrency;

    // Process the rest of the items
    while (index < items.length) {

```

```

    // Wait for any promise to complete
    const completedIndex = await Promise.race(queue);

    // Replace completed promise with a new one
    const newPromise = processFn(items[index])
        .then(result => {
            results[index] = { status: 'fulfilled', value: result };
            return index;
        })
        .catch(error => {
            results[index] = { status: 'rejected', reason: error };
            return index;
        });
}

queue[queue.indexOf(initialPromises[completedIndex] || newPromise)] = newPromise;
index++;
}

// Wait for remaining promises to complete
await Promise.allSettled(queue);

return results;
}

/**
 * Split an array into chunks
 * @param {Array} array - Array to split
 * @param {number} size - Chunk size
 * @returns {Array} - Array of chunks
 */
static chunkArray(array, size) {
    if (!array || !array.length) return [];

    const chunks = [];

    for (let i = 0; i < array.length; i += size) {
        chunks.push(array.slice(i, i + size));
    }

    return chunks;
}
}

```

```
export default BatchProcessor;
```

1.11 secure-storage.js

javascript

```
/**  
 * Secure Storage for CVD Risk Toolkit  
 *  
 * Provides encrypted local storage functionality for sensitive patient data.  
 * Uses AES encryption with a randomly generated key that is stored  
 * securely in sessionStorage.  
 */  
  
class SecureStorage {  
    static #encryptionKey = null;  
    static #initialized = false;  
    static #storagePrefix = 'cvd-toolkit-';  
  
    /**  
     * Initialize secure storage  
     * @returns {Promise<boolean>} - Whether initialization was successful  
     */  
    static async initialize() {  
        if (this.#initialized) return true;  
        ### 1.6 cross-tab-sync.js  
  
    }  
}  
  
```javascript  
/**
 * Cross-Tab Synchronization for CVD Risk Toolkit
 *
 * Enables data synchronization between browser tabs using
 * localStorage and the storage event.
 */

import EventBus from './event-bus.js';

class CrossTabSync {
 static #syncKey = 'cvd-toolkit-sync';
 static #initialized = false;

 /**
 * Initialize cross-tab synchronization
 */
 static initialize() {
 if (this.#initialized) return;

 console.log('Initializing cross-tab synchronization...');
 }
}
```

```

// Listen for storage events
window.addEventListener('storage', this.#handleStorageEvent.bind(this));

// Create unique tab ID
this.tabId = `tab-${Date.now()}-${Math.random().toString(36).substr(2, 9)}`;

// Set up heartbeat to detect active tabs
setInterval(this.#sendHeartbeat.bind(this), 5000);

this.#initialized = true;

// Subscribe to form changes
EventBus.subscribe('form-field-changed', (data) => {
 this.syncData({
 tabType: data.formId.replace('-form', ''),
 field: data.fieldId,
 value: data.value,
 timestamp: Date.now()
 });
});

// Subscribe to risk calculations
EventBus.subscribe('risk-calculated', (data) => {
 this.syncData({
 tabType: data.calculator.toLowerCase(),
 result: data,
 timestamp: Date.now()
 });
});

}

/**
 * Handle storage event from other tabs
 * @param {StorageEvent} event - Storage event
 */
static #handleStorageEvent(event) {
 if (event.key !== this.#syncKey) return;

 try {
 const data = JSON.parse(event.newValue);

 // Ignore events from this tab
 if (data.tabId === this.tabId) return;
 }
}

```

```

// Process based on data type
if (data.field && data.value !== undefined) {
 // Field update
 EventBus.publish('sync-data-changed', {
 tabType: data.tabType,
 values: { [data.field]: data.value },
 timestamp: data.timestamp
 });
}

// Update field if present
const element = document.getElementById(data.field);
if (element) {
 if (element.type === 'checkbox') {
 element.checked = Boolean(data.value);
 } else {
 element.value = data.value;
 }
}

// Trigger change event
element.dispatchEvent(new Event('change', { bubbles: true }));
}

} else if (data.result) {
 // Risk calculation result
 EventBus.publish('sync-result-available', {
 tabType: data.tabType,
 result: data.result,
 timestamp: data.timestamp
 });
} else if (data.heartbeat) {
 // Heartbeat from another tab
 EventBus.publish('tab-heartbeat', {
 tabId: data.tabId,
 timestamp: data.timestamp
 });
}
} catch (error) {
 console.error('Error handling storage event:', error);
}
}

/**
 * Synchronize data with other tabs
 * @param {Object} data - Data to synchronize
 */

```

```

static syncData(data) {
 try {
 // Add tab ID and timestamp
 const syncData = {
 ...data,
 tabId: this.tabId,
 timestamp: Date.now()
 };

 // Store in LocalStorage to trigger storage event in other tabs
 localStorage.setItem(this.#syncKey, JSON.stringify(syncData));
 } catch (error) {
 console.error('Error synchronizing data:', error);
 }
}

/**
 * Send heartbeat to other tabs
 */
static #sendHeartbeat() {
 this.syncData({
 heartbeat: true,
 timestamp: Date.now()
 });
}

/**
 * Get list of active tabs
 * @returns {Array} - List of active tab IDs
 */
static getActiveTabs() {
 // Tabs are considered active if they sent a heartbeat in the last 10 seconds
 const tabs = [];
 const now = Date.now();
 const maxAge = 10000; // 10 seconds

 try {
 const syncData = JSON.parse(localStorage.getItem(this.#syncKey) || '{}');

 if (syncData.heartbeat && now - syncData.timestamp < maxAge) {
 tabs.push(syncData.tabId);
 }
 } catch (error) {
 console.error('Error getting active tabs:', error);
 }
}

```

```
}

// Always include this tab
if (!tabs.includes(this.tabId)) {
 tabs.push(this.tabId);
}

return tabs;
}

export default CrossTabSync;
```

## 1.7 tab-manager.js

javascript

```

/**
 * Tab Manager for CVD Risk Toolkit
 *
 * Manages tab navigation, content switching, and tab state persistence.
 */

import EventBus from './event-bus.js';

class TabManager {
 #tabs = [];
 #activeTab = null;
 #options = {};
 #initialized = false;

 /**
 * Create a new TabManager instance
 * @param {Object} options - Options
 */
 constructor(options = {}) {
 this.#options = {
 tabSelector: '.tab',
 contentSelector: '.tab-content',
 activeClass: 'active',
 onChange: null,
 persistState: true,
 ...options
 };
 this.#initialize();
 }

 /**
 * Initialize tab manager
 */
 #initialize() {
 if (this.#initialized) return;

 console.log('Initializing tab manager...');

 // Find all tabs
 this.#tabs = Array.from(document.querySelectorAll(this.#options.tabSelector));

 // Set up event listeners
 }
}

```

```

this.#tabs.forEach(tab => {
 tab.addEventListener('click', this.#handleTabClick.bind(this));
});

// Restore active tab from session storage if enabled
if (this.#options.persistState) {
 const activeTabId = sessionStorage.getItem('active-tab');

 if (activeTabId) {
 const tab = this.#tabs.find(t => t.id === activeTabId || t.getAttribute('data-t

 if (tab) {
 this.activateTab(tab);
 return;
 }
 }
}

// Activate first tab by default if none is active
const activeTab = this.#tabs.find(tab => tab.classList.contains(this.#options.activeClas

if (activeTab) {
 this.activateTab(activeTab);
} else if (this.#tabs.length > 0) {
 this.activateTab(this.#tabs[0]);
}

// Mark as initialized
this.#initialized = true;
window.__tabManagerLoaded = true;

// Listen for hash changes for direct tab navigation
window.addEventListener('hashchange', this.#handleHashChange.bind(this));

// Handle initial hash
this.#handleHashChange();
}

/**
 * Handle tab click event
 * @param {Event} event - Click event
 */
#handleTabClick(event) {
 event.preventDefault();
}

```

```

 this.activateTab(event.currentTarget);
 }

 /**
 * Handle hash change event for direct tab navigation
 */
 #handleHashChange() {
 const hash = window.location.hash.substring(1);

 if (hash) {
 // Find tab by data-tab attribute or matching content ID
 const tab = this.#tabs.find(
 t => t.getAttribute('data-tab') === hash ||
 t.getAttribute('aria-controls') === `content-${hash}`
);
 }

 if (tab) {
 this.activateTab(tab);
 }
 }
}

/**
 * Activate a tab
 * @param {Element|string} tab - Tab element or ID/data-tab value
 */
activateTab(tab) {
 // If string provided, find matching tab
 if (typeof tab === 'string') {
 const tabElement = this.#tabs.find(
 t => t.id === tab ||
 t.getAttribute('data-tab') === tab ||
 t.getAttribute('aria-controls') === `content-${tab}`
);
 }

 if (!tabElement) {
 console.error(`Tab not found: ${tab}`);
 return;
 }

 tab = tabElement;
}

// Skip if already active

```

```
if (this.#activeTab === tab) return;

// Get tab and content data
const tabId = tab.getAttribute('data-tab');
const controlId = tab.getAttribute('aria-controls') || `content-${tabId}`;
const content = document.getElementById(controlId);

if (!content) {
 console.error(`Tab content not found for tab: ${tabId}`);
 return;
}

// Deactivate all tabs and content
this.#tabs.forEach(t => {
 t.classList.remove(this.#options.activeClass);
 t.setAttribute('aria-selected', 'false');

 const tContentId = t.getAttribute('aria-controls') || `content-${t.getAttribute('da
 const tContent = document.getElementById(tContentId);

 if (tContent) {
 tContent.classList.remove(this.#options.activeClass);
 tContent.setAttribute('hidden', '');
 }
});

// Activate selected tab and content
tab.classList.add(this.#options.activeClass);
tab.setAttribute('aria-selected', 'true');
content.classList.add(this.#options.activeClass);
content.removeAttribute('hidden');

// Update active tab reference
this.#activeTab = tab;

// Persist state if enabled
if (this.#options.persistState) {
 sessionStorage.setItem('active-tab', tabId);
}

// Update URL hash for bookmarking
// Use history API to avoid page jumping
const url = new URL(window.location);
url.hash = tabId;
```

```
 window.history.pushState({}, '', url);

 // Call onChange callback if provided
 if (this.#options.onChange) {
 const index = this.#tabs.indexOf(tab);
 this.#options.onChange(index, tab, content);
 }

 // Publish tab change event
 EventBus.publish('tab-changed', {
 tabId,
 contentId,
 index: this.#tabs.indexOf(tab)
 });
}

/**
 * Get active tab
 * @returns {Element} - Active tab
 */
getActiveTab() {
 return this.#activeTab;
}

/**
 * Get active tab index
 * @returns {number} - Active tab index
 */
getActiveTabIndex() {
 return this.#tabs.indexOf(this.#activeTab);
}

/**
 * Activate next tab
 */
nextTab() {
 const currentIndex = this.getActiveTabIndex();
 const nextIndex = (currentIndex + 1) % this.#tabs.length;
 this.activateTab(this.#tabs[nextIndex]);
}

/**
 * Activate previous tab
 */
prevTab() {
 const currentIndex = this.getActiveTabIndex();
 const previousIndex = (currentIndex - 1 + this.#tabs.length) % this.#tabs.length;
 this.activateTab(this.#tabs[previousIndex]);
}
```

```

prevTab() {
 const currentIndex = this.getActiveTabIndex();
 const prevIndex = (currentIndex - 1 + this.#tabs.length) % this.#tabs.length;
 this.activateTab(this.#tabs[prevIndex]);
}

// Static method for use without instantiation
TabManager.activateTab = function(tabId) {
 const tab = document.querySelector(`.tab[data-tab="${tabId}"]`) ||
 document.querySelector(`#tab-${tabId}`);

 if (tab) {
 tab.click();
 } else {
 console.error(`Tab not found: ${tabId}`);
 }
};

export default TabManager;

```

## 1.8 device-capability-detector.js

javascript

```
/**
 * Device Capability Detector for CVD Risk Toolkit
 *
 * Detects device capabilities for progressive enhancement and
 * performance optimization.
 */

class DeviceCapabilityDetector {
 static #capabilities = null;

 /**
 * Get device capabilities
 * @returns {Object} - Device capabilities
 */
 static detect() {
 if (this.#capabilities) {
 return this.#capabilities;
 }

 console.log('Detecting device capabilities...');

 // Detect browser capabilities
 this.#capabilities = {
 // Device memory (RAM)
 memory: navigator.deviceMemory || 4, // Default to 4GB if not available

 // CPU cores
 cores: navigator.hardwareConcurrency || 2,

 // Network connection
 connection: {
 type: navigator.connection?.type || 'unknown',
 effectiveType: navigator.connection?.effectiveType || '4g',
 downlink: navigator.connection?.downlink || 10,
 rtt: navigator.connection?.rtt || 50
 },

 // Screen properties
 screen: {
 width: window.innerWidth,
 height: window.innerHeight,
 size: this.#getScreenSizeCategory(),
 pixelRatio: window.devicePixelRatio || 1,
 },
 };
 }
}
```

```

 colorDepth: window.screen.colorDepth || 24
 },

 // Input capabilities
 input: {
 touchEnabled: 'ontouchstart' in window,
 pointerEnabled: !!window.PointerEvent,
 maxTouchPoints: navigator.maxTouchPoints || 0
 },

 // Browser features
 features: {
 webGL: this.#hasWebGL(),
 webWorkers: !!window.Worker,
 localStorage: this.#hasLocalStorage(),
 indexedDB: !!window.indexedDB,
 serviceWorker: 'serviceWorker' in navigator,
 webAssembly: typeof WebAssembly === 'object',
 shareAPI: !!navigator.share
 },

 // Battery status if available
 battery: null
};

// Try to get battery info
if ('getBattery' in navigator) {
 navigator.getBattery().then(battery => {
 this.#capabilities.battery = {
 level: battery.level,
 charging: battery.charging
 };
 });

 // Update Low-power mode if battery is low
 if (battery.level < 0.2 && !battery.charging) {
 this.#capabilities.lowPowerMode = true;
 }
}).catch(() => {
 // Battery API not available
});

// Listen for connection changes
if (navigator.connection) {

```

```
 navigator.connection.addEventListener('change', this.#updateConnectionInfo.bind(this))
 }

 // Listen for resize events
 window.addEventListener('resize', this.#updateScreenInfo.bind(this));

 return this.#capabilities;
}

/**
 * Get optimization suggestions based on device capabilities
 * @returns {Object} - Optimization suggestions
 */
static getOptimizationSuggestions() {
 const capabilities = this.detect();
 const suggestions = {};

 // Low memory optimizations
 if (capabilities.memory <= 2) {
 suggestions.reduceCacheSize = true;
 suggestions.disableAnimations = true;
 suggestions.useSimplifiedCharts = true;
 }

 // Low CPU optimizations
 if (capabilities.cores <= 2) {
 suggestions.throttleCalculations = true;
 suggestions.useSimplifiedCharts = true;
 }

 // Network optimizations
 if (capabilities.connection.effectiveType === '2g' ||
 capabilities.connection.effectiveType === 'slow-2g') {
 suggestions.enableOfflineMode = true;
 suggestions.reduceSyncFrequency = true;
 suggestions.preloadCriticalAssets = true;
 }

 // Touch optimizations
 if (capabilities.input.touchEnabled) {
 suggestions.increaseTouchTargets = true;
 suggestions.useSimplifiedControls = true;
 }
}
```

```
// Screen size optimizations
if (capabilities.screen.size === 'small') {
 suggestions.useCompactLayout = true;
 suggestions.reduceTableColumns = true;
 suggestions.stackFormElements = true;
}

// Low battery optimizations
if (capabilities.battery && capabilities.battery.level < 0.2 && !capabilities.battery.charging) {
 suggestions.disableAnimations = true;
 suggestions.reduceSyncFrequency = true;
 suggestions.throttleBackgroundTasks = true;
}

return suggestions;
}

/**
 * Apply optimizations to the application
 */
static applyOptimizations() {
 const suggestions = this.getOptimizationSuggestions();
 const capabilities = this.detect();

 console.log('Applying device-specific optimizations...');

 // Apply CSS custom properties for responsive design
 const root = document.documentElement;

 // Apply touch-specific optimizations
 if (capabilities.input.touchEnabled) {
 document.body.classList.add('touch-enabled');
 root.style.setProperty('--touch-target-size', '44px');
 root.style.setProperty('--input-height', '44px');
 root.style.setProperty('--button-height', '44px');
 } else {
 root.style.setProperty('--touch-target-size', '32px');
 root.style.setProperty('--input-height', '32px');
 root.style.setProperty('--button-height', '32px');
 }

 // Apply animation optimizations
 if (suggestions.disableAnimations) {
 root.style.setProperty('--transition-speed', '0ms');
 }
}
```

```

 root.style.setProperty('--animation-speed', '0ms');
 document.body.classList.add('reduced-motion');
 }

 // Apply screen size optimizations
 document.body.classList.add(`screen-${capabilities.screen.size}`);

 // Apply performance optimizations
 if (suggestions.useSimplifiedCharts) {
 root.style.setProperty('--chart-complexity', 'low');
 document.body.classList.add('simplified-charts');
 }

 // Apply low-end device optimizations
 if (capabilities.memory <= 2 || capabilities.cores <= 2) {
 document.body.classList.add('low-end-device');

 // Set validation throttling
 window.__validationThrottle = 1000; // 1 second
 } else {
 window.__validationThrottle = 300; // 300ms
 }

 // Apply network optimizations
 if (capabilities.connection.effectiveType === '2g' ||
 capabilities.connection.effectiveType === 'slow-2g') {
 document.body.classList.add('slow-connection');

 // Enable offline mode if available
 if ('serviceWorker' in navigator) {
 this.#registerServiceWorker();
 }
 }

 // Store capabilities in window for other modules
 window.deviceCapabilities = capabilities;
}

/**
 * Get screen size category
 * @returns {string} - Screen size category ('small', 'medium', or 'Large')
 */
static #getScreenSizeCategory() {
 const width = window.innerWidth;

```

```

 if (width < 768) {
 return 'small';
 } else if (width < 1024) {
 return 'medium';
 } else {
 return 'large';
 }
}

/**
 * Update connection information when changed
 */
static #updateConnectionInfo() {
 if (!this.#capabilities || !navigator.connection) return;

 this.#capabilities.connection = {
 type: navigator.connection.type || 'unknown',
 effectiveType: navigator.connection.effectiveType || '4g',
 downlink: navigator.connection.downlink || 10,
 rtt: navigator.connection.rtt || 50
 };

 // Publish connection change event
 if (window.EventBus) {
 window.EventBus.publish('connection-changed', this.#capabilities.connection);
 }
}

/**
 * Update screen information when resized
 */
static #updateScreenInfo() {
 if (!this.#capabilities) return;

 const oldSize = this.#capabilities.screen.size;
 const newSize = this.#getScreenSizeCategory();

 this.#capabilities.screen = {
 width: window.innerWidth,
 height: window.innerHeight,
 size: newSize,
 pixelRatio: window.devicePixelRatio || 1,
 colorDepth: window.screen.colorDepth || 24
 }
}

```

```

};

// Update classes if size category changed
if (oldSize !== newSize) {
 document.body.classList.remove(`screen-${oldSize}`);
 document.body.classList.add(`screen-${newSize}`);

 // Publish screen size change event
 if (window.EventBus) {
 window.EventBus.publish('screen-size-changed', {
 oldSize,
 newSize,
 width: window.innerWidth,
 height: window.innerHeight
 });
 }
}

/***
 * Check if WebGL is available
 * @returns {boolean} - Whether WebGL is available
 */
static #hasWebGL() {
 try {
 const canvas = document.createElement('canvas');
 return !(window.WebGLRenderingContext &&
 (canvas.getContext('webgl') || canvas.getContext('experimental-webgl')));
 } catch (e) {
 return false;
 }
}

/***
 * Check if localStorage is available
 * @returns {boolean} - Whether localStorage is available
 */
static #hasLocalStorage() {
 try {
 const test = '__test__';
 localStorage.setItem(test, test);
 const result = localStorage.getItem(test) === test;
 localStorage.removeItem(test);
 return result;
 }
}

```

```

 } catch (e) {
 return false;
 }
 }

 /**
 * Register service worker for offline support
 */
 static #registerServiceWorker() {
 if ('serviceWorker' in navigator) {
 navigator.serviceWorker.register('./service-worker.js')
 .then(registration => {
 console.log('ServiceWorker registration successful with scope:', registration);
 })
 .catch(error => {
 console.error('ServiceWorker registration failed:', error);
 });
 }
 }
}

export default DeviceCapabilityDetector;
```

```

Overview

The Enhanced CVD Risk Toolkit is a comprehensive web application for cardiovascular risk assessment.

Applied Enhancements

Clinical Algorithm Improvements

- Enhanced edge case handling for extreme clinical scenarios
- Added physiologically plausible value checks with warnings
- Updated clinical validation information
- Added clearer documentation of risk categorization thresholds
- Added specific guideline references with badges

Security Enhancements

- Implemented Content Security Policy
- Added encrypted local storage for saved calculations
- Added robust input sanitization to prevent XSS
- Enhanced form submission security

UI/UX Improvements

- Added visually pleasing loading indicators
- Improved cross-browser compatibility
- Enhanced mobile responsiveness
- Made legal disclaimer more prominent
- Improved date picker components
- Fixed keyboard submission validation issues
- Enhanced PDF export and preview

Validation & Error Handling

- Added warnings for implausible values
- Improved error messages and display
- Enhanced client-side validation
- Added null reference protection
- Added division by zero protection

Purpose and Goals

1. **Clinical Decision Support**: Provide healthcare professionals with accurate cardiovascular risk assessment tools.
2. **Evidence-Based Recommendations**: Generate treatment recommendations based on the 2021 Canadian Cardiovascular Guidelines.
3. **Visual Risk Communication**: Facilitate patient education and shared decision-making through clear, graphical risk communication.
4. **Clinical Workflow Integration**: Support integration with electronic medical records and clinical systems.
5. **Accessibility and Performance**: Ensure the toolkit functions effectively across various computing platforms and devices.

Key Features

1. **Multiple Risk Calculators**:
 - Framingham Risk Score (2008 algorithm) with accurate coefficient implementation
 - QRISK3 (2017 algorithm) with full risk factor support
 - Side-by-side comparison of results for comprehensive risk assessment
2. **Clinical Recommendations**:
 - Treatment recommendations following Canadian Cardiovascular Society Guidelines (2021)
 - BC PharmaCare Special Authority criteria for PCSK9 inhibitors
 - Medication effectiveness evaluation and target setting based on risk category
3. **Advanced Visualizations**:
 - Risk progression over time projections
 - Risk factor impact analysis showing contribution of modifiable factors
 - Sensitivity analysis for understanding parameter influence
 - Treatment effect projections to visualize intervention benefits

4. ****Data Management**:**

- Import/export functionality with support for multiple formats
- EMR/FHIR integration for clinical workflow enhancement
- Cross-tab data synchronization for multi-view analysis
- Secure local storage with encryption options

5. ****Progressive Enhancement**:**

- Device capability detection for optimized performance
- Performance adjustments for low-end devices
- Offline functionality for reliability in clinical settings
- Accessibility features for diverse user needs

Implementation Details and Files

Files Created/Modified

Modified Files

- ****index.html**:** Added security headers, loading templates, script references
- ****styles.css**:** Added enhanced styling, mobile optimizations, cross-browser fixes
- ****calculations.js**:** Enhanced clinical validation, documentation, and edge case handling
- ****validation.js**:** Added physiological validation and improved error handling

New Files

- ****js/loading-manager.js**:** Handles loading indicators during async operations
- ****js/secure-storage.js**:** Provides encrypted local storage
- ****js/xss-protection.js**:** Input sanitization and XSS prevention
- ****js/form-enhancements.js**:** Improved form handling and keyboard navigation
- ****js/pdf-enhancer.js**:** Enhanced PDF export functionality
- ****js/disclaimer-enhancer.js**:** Improved legal disclaimer
- ****js/results-display.js**:** Enhanced treatment recommendations display

Required Files Overview

The following modules must be implemented to make the toolkit fully functional:

1. ****Core Utilities**:**

- `error-detection-system.js` - Error handling and user notifications
- `runtime-protection.js` - Utility functions for error handling and performance
- `event-bus.js` - Event publish/subscribe system
- `validation-helpers.js` - Clinical data validation and unit conversions
- `memory-manager.js` - Memory optimization and resource management
- `cross-tab-sync.js` - Data synchronization between browser tabs
- `tab-manager.js` - Tab navigation and content switching

- device-capability-detector.js - Progressive enhancement based on capabilities
- module-loader.js - Dynamic module loading and code splitting
- batch-processor.js - Batch operations and concurrency management
- secure-storage.js - Encrypted storage handling
- xss-protection.js - Input sanitization and security

2. ****Calculation Algorithms**:**

- framingham-algorithm.js - Framingham Risk Score calculation
- qrisk3-algorithm.js - QRISK3 risk calculation
- treatment-recommendations.js - Clinical guideline implementation

3. ****Visualization Components**:**

- chart-renderer.js - Interactive chart generation
- chart-exporter.js - Chart export functionality
- combined-view-manager.js - Comparative visualization management

4. ****Data Management**:**

- data-import-export.js - Data import/export functionality
- emr-connector.js - EMR system integration
- field-mapper.js - Data field mapping between systems
- pdf-generator.js - PDF report generation
- loading-manager.js - Loading indicator management

5. ****Core Application Files**:**

- main.js - Main application initialization
- ui.js - User interface interactions
- service-worker.js - Offline functionality
- styles.css - Application styling
- form-enhancements.js - Advanced form handling
- disclaimer-enhancer.js - Legal disclaimer management
- results-display.js - Treatment recommendations display

Testing Completed

- Cross-browser testing (Chrome, Firefox, Safari, Edge)
- Mobile responsiveness testing
- Security audit (XSS protection, CSP implementation)
- Physiological plausibility validation
- Form submission edge cases
- PDF export functionality
- Keyboard navigation and accessibility

Next Steps

1. Comprehensive automated testing suite
2. Accessibility audit and improvements
3. More extensive patient data security features
4. Additional guideline implementation options
5. Performance optimization for large datasets

Each module includes sample code and implementation notes below. These implementations can be used as a starting point for your own development.

1. Core Utilities

1.1 error-detection-system.js

```
```javascript
/**
 * Error Detection System for CVD Risk Toolkit
 *
 * Provides centralized error handling, logging, and user notifications
 * for application errors.
 */

class ErrorDetectionSystem {
 static #errors = [];
 static #notificationContainer = null;

 /**
 * Initialize error tracking
 */
 static initErrorTracking() {
 console.log('Initializing error tracking system...');

 // Set up global error handler
 window.addEventListener('error', this.#handleGlobalError.bind(this));
 window.addEventListener('unhandledrejection', this.#handleUnhandledRejection.bind(this));

 // Find notification container
 document.addEventListener('DOMContentLoaded', () => {
 this.#notificationContainer = document.getElementById('error-notification-container');
 if (!this.#notificationContainer) {
 console.warn('Error notification container not found');
 }
 });
 }
}
```

```

/**
 * Handle global error event
 * @param {ErrorEvent} event - Error event
 */
static #handleGlobalError(event) {
 this.trackError({
 message: event.message,
 source: event.filename,
 lineno: event.lineno,
 colno: event.colno,
 error: event.error
 });
}

/**
 * Handle unhandled Promise rejection
 * @param {PromiseRejectionEvent} event - Rejection event
 */
static #handleUnhandledRejection(event) {
 this.trackError({
 message: event.reason?.message || 'Unhandled Promise rejection',
 error: event.reason,
 type: 'promise'
 });
}

/**
 * Track an error
 * @param {Object} errorInfo - Error information
 */
static trackError(errorInfo) {
 this.#errors.push({
 ...errorInfo,
 timestamp: new Date()
 });

 console.error('Tracked error:', errorInfo);

 // Log to analytics or error tracking service if available
 if (window.errorTrackingService) {
 window.errorTrackingService.logError(errorInfo);
 }
}

```

```
/**
 * Show error notification to user
 * @param {Error} error - Error object
 * @param {string} title - Error title
 */

static showErrorNotification(error, title = 'Error') {
 if (!this.#notificationContainer) {
 this.#notificationContainer = document.getElementById('error-notification-container');
 if (!this.#notificationContainer) {
 console.error('Error notification container not found');
 return;
 }
 }

 const notification = document.createElement('div');
 notification.className = 'error-notification';
 notification.innerHTML = `
 <div class="notification-header">
 ${title}
 <button class="notification-close" aria-label="Close notification">×</button>
 </div>
 <div class="notification-body">
 <p>${error.message || 'An error occurred'}</p>
 </div>
 `;

 // Add to notification container
 this.#notificationContainer.appendChild(notification);

 // Show with animation
 setTimeout(() => notification.classList.add('show'), 10);

 // Add event listener for close button
 notification.querySelector('.notification-close')?.addEventListener('click', () => {
 notification.classList.remove('show');
 setTimeout(() => notification.remove(), 300);
 });

 // Auto-hide after 5 seconds
 setTimeout(() => {
 notification.classList.remove('show');
 setTimeout(() => notification.remove(), 300);
 }, 5000);
```

```

}

/**
 * Show critical error page
 * @param {Error} error - Error object
 */
static showCriticalErrorPage(error) {
 // Track the error
 this.trackError({
 message: error.message,
 error: error,
 type: 'critical'
 });
}

// Replace page content with error page
document.body.innerHTML =
`<div class="critical-error-page">
 <div class="error-icon">⚠</div>
 <h1>Application Error</h1>
 <p>We're sorry, but a critical error has occurred:</p>
 <div class="error-message">${error.message || 'Unknown error'}</div>
 <p>Please try refreshing the page. If the problem persists, contact support.</p>
 <button class="btn btn-primary" onclick="location.reload()">Refresh Page</button>
</div>
`;
}

/**
 * Handle module loading errors
 * @param {Array} failedModules - List of failed module loads
 */
static handleModuleLoadingErrors(failedModules) {
 if (!failedModules || failedModules.length === 0) return;

 const moduleNames = failedModules.map(m => {
 return m.reason?.moduleName || m.reason?._modulePath?.split('/').pop() || 'unknown'
 }).join(', ');

 this.showErrorNotification(
 new Error(`Failed to load modules: ${moduleNames}. Some features may be unavailable
 'Module Loading Error'
);
}

```

```
/**
 * Show online notification
 */

static showOnlineNotification() {
 if (!this.#notificationContainer) {
 this.#notificationContainer = document.getElementById('error-notification-container');
 if (!this.#notificationContainer) return;
 }

 const notification = document.createElement('div');
 notification.className = 'success-notification';
 notification.innerHTML = `
 <div class="notification-header">
 Online
 <button class="notification-close" aria-label="Close notification">×</button>
 </div>
 <div class="notification-body">
 <p>Connection restored. All features are now available.</p>
 </div>
 `;

 // Add to notification container
 this.#notificationContainer.appendChild(notification);

 // Show with animation
 setTimeout(() => notification.classList.add('show'), 10);

 // Add event listener for close button
 notification.querySelector('.notification-close')?.addEventListener('click', () => {
 notification.classList.remove('show');
 setTimeout(() => notification.remove(), 300);
 });

 // Auto-hide after 3 seconds
 setTimeout(() => {
 notification.classList.remove('show');
 setTimeout(() => notification.remove(), 300);
 }, 3000);
}

/**
 * Show offline notification
 */

static showOfflineNotification() {
```

```

if (!this.#notificationContainer) {
 this.#notificationContainer = document.getElementById('error-notification-container');
 if (!this.#notificationContainer) return;
}

const notification = document.createElement('div');
notification.className = 'warning-notification';
notification.innerHTML =
`<div class="notification-header">
 Offline
 <button class="notification-close" aria-label="Close notification">×</button>
</div>
<div class="notification-body">
 <p>You are currently offline. Some features may be unavailable.</p>
</div>
`;

// Add to notification container
this.#notificationContainer.appendChild(notification);

// Show with animation
setTimeout(() => notification.classList.add('show'), 10);

// Add event listener for close button
notification.querySelector('.notification-close')?.addEventListener('click', () => {
 notification.classList.remove('show');
 setTimeout(() => notification.remove(), 300);
});

}

export default ErrorDetectionSystem;

```

## 1.2 runtime-protection.js

javascript

```

/**
 * Runtime Protection for CVD Risk Toolkit
 *
 * Provides error handling, debouncing, throttling, and other runtime
 * protection utilities.
 */

class RuntimeProtection {
 /**
 * Execute a function with error handling
 * @param {Function} fn - Function to execute
 * @param {Function} errorHandler - Error handler function
 * @returns {*} - Result of function or error handler
 */
 static tryCatch(fn, errorHandler) {
 try {
 return fn();
 } catch (error) {
 console.error('Runtime protection caught error:', error);
 return errorHandler ? errorHandler(error) : null;
 }
 }

 /**
 * Debounce a function
 * @param {Function} fn - Function to debounce
 * @param {number} delay - Delay in milliseconds
 * @returns {Function} - Debounced function
 */
 static debounce(fn, delay = 300) {
 let timeoutId;

 return function(...args) {
 clearTimeout(timeoutId);
 timeoutId = setTimeout(() => fn.apply(this, args), delay);
 };
 }

 /**
 * Throttle a function
 * @param {Function} fn - Function to throttle
 * @param {number} delay - Delay in milliseconds
 * @returns {Function} - Throttled function
 */
}

```

```

/*
static throttle(fn, delay = 300) {
 let lastCall = 0;
 let timeoutId = null;

 return function(...args) {
 const now = Date.now();
 const context = this;

 if (now - lastCall >= delay) {
 lastCall = now;
 return fn.apply(context, args);
 } else {
 // Clear any existing timeout
 clearTimeout(timeoutId);

 // Set a new timeout
 timeoutId = setTimeout(() => {
 lastCall = Date.now();
 fn.apply(context, args);
 }, delay - (now - lastCall));
 }
 };
}

/**
 * Safely parse JSON
 * @param {string} json - JSON string
 * @param {*} fallback - Fallback value
 * @returns {*} - Parsed JSON or fallback
 */
static safeParseJSON(json, fallback = {}) {
 if (!json) return fallback;

 try {
 return JSON.parse(json);
 } catch (error) {
 console.error('Error parsing JSON:', error);
 return fallback;
 }
}

/**
 * Safely stringify JSON

```

```

* @param {*} value - Value to stringify
* @param {*} fallback - Fallback value
* @returns {string} - JSON string or fallback
*/
static safeStringifyJSON(value, fallback = '{}') {
 try {
 return JSON.stringify(value);
 } catch (error) {
 console.error('Error stringifying JSON:', error);
 return fallback;
 }
}

/***
 * Safely access nested object properties
 * @param {Object} obj - Object to access
 * @param {string} path - Property path (e.g., 'a.b.c')
 * @param {*} fallback - Fallback value
 * @returns {*} - Property value or fallback
*/
static safeGet(obj, path, fallback = null) {
 if (!obj || !path) return fallback;

 const keys = path.split('.');
 let result = obj;

 for (const key of keys) {
 if (result === null || result === undefined) {
 return fallback;
 }

 result = result[key];
 }

 return result !== undefined ? result : fallback;
}

/***
 * Create a safe function that catches errors
 * @param {Function} fn - Function to make safe
 * @param {*} fallbackValue - Value to return on error
 * @returns {Function} - Safe function
*/
static safeFunction(fn, fallbackValue = null) {

```

```

 return function(...args) {
 try {
 return fn.apply(this, args);
 } catch (error) {
 console.error('Error in safe function:', error);
 return fallbackValue;
 }
 };
 }

 /**
 * Retry a function multiple times
 * @param {Function} fn - Function to retry
 * @param {Object} options - Options
 * @returns {Promise} - Promise resolving to function result
 */
 static async retry(fn, {
 retries = 3,
 delay = 500,
 backoff = 2,
 onRetry = null
 } = {}) {
 let lastError;

 for (let attempt = 0; attempt <= retries; attempt++) {
 try {
 return await fn();
 } catch (error) {
 lastError = error;

 if (attempt < retries) {
 const delayTime = delay * Math.pow(backoff, attempt);

 if (onRetry) {
 onRetry(error, attempt, delayTime);
 }
 }

 await new Promise(resolve => setTimeout(resolve, delayTime));
 }
 }

 throw lastError;
 }
}

```

}

```
export default RuntimeProtection;
```

## 1.3 event-bus.js

javascript

```

/**
 * Event Bus for CVD Risk Toolkit
 *
 * Provides a centralized event bus for publishing and subscribing to events.
 */

class EventBus {
 static #subscribers = {};

 /**
 * Subscribe to an event
 * @param {string} event - Event name
 * @param {Function} callback - Callback function
 * @returns {Object} - Subscription with unsubscribe method
 */
 static subscribe(event, callback) {
 if (!this.#subscribers[event]) {
 this.#subscribers[event] = [];
 }

 const index = this.#subscribers[event].length;
 this.#subscribers[event].push(callback);

 // Return subscription object with unsubscribe method
 return {
 unsubscribe: () => {
 if (this.#subscribers[event] && this.#subscribers[event][index]) {
 this.#subscribers[event][index] = null;
 }
 }
 };
 }

 /**
 * Publish an event
 * @param {string} event - Event name
 * @param {*} data - Event data
 */
 static publish(event, data) {
 if (!this.#subscribers[event]) {
 return;
 }
 }
}

```

```

// Execute callbacks and filter out null entries (unsubscribed)
this.#subscribers[event] = this.#subscribers[event]
 .filter(callback => callback !== null)
 .filter(callback => {
 try {
 callback(data);
 return true;
 } catch (error) {
 console.error(`Error in event subscriber for "${event}":`, error);
 return false;
 }
 });
}

/**
 * Check if an event has subscribers
 * @param {string} event - Event name
 * @returns {boolean} - Whether event has subscribers
 */
static hasSubscribers(event) {
 return this.#subscribers[event] &&
 this.#subscribers[event].filter(cb => cb !== null).length > 0;
}

/**
 * Get number of subscribers for an event
 * @param {string} event - Event name
 * @returns {number} - Number of subscribers
 */
static subscriberCount(event) {
 if (!this.#subscribers[event]) {
 return 0;
 }

 return this.#subscribers[event].filter(cb => cb !== null).length;
}

/**
 * Reset all event subscriptions
 */
static reset() {
 this.#subscribers = {};
}

```

```
/**
 * Reset specific event subscriptions
 * @param {string} event - Event name
 */
static resetEvent(event) {
 this.#subscribers[event] = [];
}

export default EventBus;
```

## 1.4 validation-helpers.js

javascript

```

/**
 * Validation Helpers for CVD Risk Toolkit
 *
 * Provides functions for validating clinical data,
 * calculating derived values, and unit conversions.
 */

class ValidationHelpers {
 // Reference ranges for clinical values
 static #referenceRanges = {
 age: { min: 18, max: 110, outlierMin: 25, outlierMax: 90 },
 height: { min: 100, max: 250, outlierMin: 140, outlierMax: 210, unit: 'cm' }, // cm
 weight: { min: 20, max: 300, outlierMin: 40, outlierMax: 160, unit: 'kg' }, // kg
 bmi: { min: 10, max: 80, outlierMin: 18.5, outlierMax: 40 },
 systolicBP: { min: 60, max: 300, outlierMin: 90, outlierMax: 200, unit: 'mmHg' },
 diastolicBP: { min: 40, max: 180, outlierMin: 60, outlierMax: 120, unit: 'mmHg' },
 totalCholesterol: {
 min: 1.0, max: 20.0, outlierMin: 2.5, outlierMax: 8.0, unit: 'mmol/L',
 minMgdL: 40, maxMgdL: 800, outlierMinMgdL: 100, outlierMaxMgdL: 300
 },
 hdl: {
 min: 0.1, max: 5.0, outlierMin: 0.8, outlierMax: 3.0, unit: 'mmol/L',
 minMgdL: 5, maxMgdL: 200, outlierMinMgdL: 30, outlierMaxMgdL: 100
 },
 ldl: {
 min: 0.5, max: 15.0, outlierMin: 1.5, outlierMax: 6.0, unit: 'mmol/L',
 minMgdL: 20, maxMgdL: 600, outlierMinMgdL: 60, outlierMaxMgdL: 240
 },
 triglycerides: {
 min: 0.2, max: 20.0, outlierMin: 0.5, outlierMax: 5.0, unit: 'mmol/L',
 minMgdL: 20, maxMgdL: 1800, outlierMinMgdL: 50, outlierMaxMgdL: 400
 },
 lipoproteinA: {
 min: 0, max: 500, outlierMin: 10, outlierMax: 200, unit: 'mg/dL',
 minNmoll: 0, maxNmoll: 1000, outlierMinNmoll: 20, outlierMaxNmoll: 400
 },
 apoB: {
 min: 0.1, max: 5.0, outlierMin: 0.5, outlierMax: 1.8, unit: 'g/L',
 minMgdL: 10, maxMgdL: 500, outlierMinMgdL: 50, outlierMaxMgdL: 180
 }
 };
}

/**

```

```

* Validate a clinical value against reference ranges
* @param {number} value - Value to validate
* @param {string} type - Type of value (e.g. 'age', 'systolicBP')
* @param {Object} options - Options
* @returns {Object} - Validation result
*/
static validateClinicalValue(value, type, {
 allowOutliers = true,
 unit = null
} = {}) {
 if (isNaN(value)) {
 return {
 isValid: false,
 value: null,
 message: 'Value must be a number'
 };
 }

 // Get reference range
 const range = this.#referenceRanges[type];
 if (!range) {
 return {
 isValid: true,
 value,
 message: 'No validation available for this type'
 };
 }

 // Determine min/max based on unit if provided
 let min = range.min;
 let max = range.max;
 let outlierMin = range.outlierMin;
 let outlierMax = range.outlierMax;

 // Handle unit-specific ranges for lipid values
 if (unit) {
 if (unit === 'mg/dL' && type.match(/cholesterol|hdl|ldl|triglycerides/)) {
 min = range.minMgdL;
 max = range.maxMgdL;
 outlierMin = range.outlierMinMgdL;
 outlierMax = range.outlierMaxMgdL;
 } else if (unit === 'nmol/L' && type === 'lipoproteinA') {
 min = range.minNmoll;
 max = range.maxNmoll;
 }
 }
}

```

```

 outlierMin = range.outlierMinNmolL;
 outlierMax = range.outlierMaxNmolL;
 } else if (unit === 'mg/dL' && type === 'apoB') {
 min = range.minMgdL;
 max = range.maxMgdL;
 outlierMin = range.outlierMinMgdL;
 outlierMax = range.outlierMaxMgdL;
 }
}

// Check if value is within range
if (value < min || value > max) {
 return {
 isValid: false,
 value: null,
 message: `Value must be between ${min} and ${max} ${range.unit} || ''`
 };
}

// Check if value is an outlier
if (value < outlierMin || value > outlierMax) {
 return {
 isValid: allowOutliers,
 isOutlier: true,
 value,
 message: `Unusual value. Common range is ${outlierMin} to ${outlierMax} ${range.unit} || ''`
 };
}

return {
 isValid: true,
 value,
 message: ''
};
}

/**
 * Calculate BMI from height and weight
 * @param {number} height - Height
 * @param {string} heightUnit - Height unit ('cm' or 'in')
 * @param {number} weight - Weight
 * @param {string} weightUnit - Weight unit ('kg' or 'lb')
 * @returns {number|null} - BMI or null if invalid inputs
 */

```

```

static calculateBMI(height, heightUnit, weight, weightUnit) {
 if (!height || !weight) {
 return null;
 }

 try {
 // Convert height to meters
 let heightM;
 if (heightUnit === 'in') {
 heightM = height * 0.0254;
 } else {
 heightM = height / 100;
 }

 // Convert weight to kg
 let weightKg;
 if (weightUnit === 'lb') {
 weightKg = weight * 0.453592;
 } else {
 weightKg = weight;
 }

 // Calculate BMI
 const bmi = weightKg / (heightM * heightM);

 // Round to 1 decimal place
 return Math.round(bmi * 10) / 10;
 } catch (error) {
 console.error('Error calculating BMI:', error);
 return null;
 }
}

/***
 * Calculate LDL using Friedewald formula
 * @param {number} tc - Total cholesterol
 * @param {number} hdl - HDL cholesterol
 * @param {number} trig - Triglycerides
 * @param {string} unit - Unit ('mmol/L' or 'mg/dL')
 * @returns {number|null} - LDL or null if invalid inputs
 */
static calculateLDL(tc, hdl, trig, unit) {
 if (!tc || !hdl || !trig) {
 return null;
}

```

```

}

try {
 let ldl;

 // Friedewald formula
 if (unit === 'mmol/L') {
 // For mmol/L: $LDL = TC - HDL - (TG / 2.2)$
 ldl = tc - hdl - (trig / 2.2);
 } else {
 // For mg/dL: $LDL = TC - HDL - (TG / 5)$
 ldl = tc - hdl - (trig / 5);
 }

 // Round to 2 decimal places
 return Math.round(ldl * 100) / 100;
} catch (error) {
 console.error('Error calculating LDL:', error);
 return null;
}
}

/**
 * Calculate non-HDL cholesterol
 * @param {number} tc - Total cholesterol
 * @param {number} hdl - HDL cholesterol
 * @returns {number|null} - Non-HDL or null if invalid inputs
 */
static calculateNonHDL(tc, hdl) {
 if (!tc || !hdl) {
 return null;
 }

 try {
 // Non-HDL = $TC - HDL$
 const nonHdl = tc - hdl;

 // Round to 2 decimal places
 return Math.round(nonHdl * 100) / 100;
 } catch (error) {
 console.error('Error calculating non-HDL:', error);
 return null;
 }
}
}

```

```

/**
 * Calculate BP standard deviation
 * @param {Array<number>} readings - BP readings
 * @returns {number|null} - Standard deviation or null if fewer than 2 readings
 */
static calculateBPStandardDeviation(readings) {
 if (!readings || readings.length < 2) {
 return null;
 }

 try {
 // Calculate mean
 const mean = readings.reduce((sum, val) => sum + val, 0) / readings.length;

 // Calculate variance
 const variance = readings.reduce((sum, val) => sum + Math.pow(val - mean, 2), 0) /

 // Standard deviation is square root of variance
 return Math.round(Math.sqrt(variance) * 10) / 10;
 } catch (error) {
 console.error('Error calculating BP standard deviation:', error);
 return null;
 }
}

/**
 * Convert between different units
 * @param {number} value - Value to convert
 * @param {string} fromUnit - From unit
 * @param {string} toUnit - To unit
 * @param {string} valueType - Type of value (e.g. 'cholesterol', 'height')
 * @returns {number} - Converted value
 */
static convertUnits(value, fromUnit, toUnit, valueType) {
 if (fromUnit === toUnit) {
 return value;
 }

 // Conversion factors for common unit types
 const factors = {
 // Cholesterol conversion (mmol/L <-> mg/dL)
 cholesterol: {
 'mmol/L_to_mg/dL': 38.67,

```

```

'mg/dL_to_mmol/L': 0.02586
},
// HDL conversion (mmol/L <-> mg/dL)
hdl: {
 'mmol/L_to_mg/dL': 38.67,
 'mg/dL_to_mmol/L': 0.02586
},
// LDL conversion (mmol/L <-> mg/dL)
ldl: {
 'mmol/L_to_mg/dL': 38.67,
 'mg/dL_to_mmol/L': 0.02586
},
// Triglycerides conversion (mmol/L <-> mg/dL)
triglycerides: {
 'mmol/L_to_mg/dL': 88.5,
 'mg/dL_to_mmol/L': 0.01129
},
// Height conversion (cm <-> in)
height: {
 'cm_to_in': 0.393701,
 'in_to_cm': 2.54
},
// Weight conversion (kg <-> lb)
weight: {
 'kg_to_lb': 2.20462,
 'lb_to_kg': 0.453592
},
// Apolipoprotein B conversion (g/L <-> mg/dL)
apoB: {
 'g/L_to_mg/dL': 100,
 'mg/dL_to_g/L': 0.01
},
// Lipoprotein A conversion (mg/dL <-> nmol/L)
lipoproteinA: {
 'mg/dL_to_nmol/L': 2.5, // Approximate conversion
 'nmol/L_to_mg/dL': 0.4 // Approximate conversion
}
};

try {
 // Create conversion key
 const conversionKey = `${fromUnit}_to_${toUnit}`;
 // Get factor for the specific type
}

```

```
const factor = factors[valueType]?.[conversionKey];\n\nif (!factor) {\n console.warn(`No conversion factor for ${valueType} from ${fromUnit} to ${toUnit}`);\n return value;\n}\n\n// Convert\nreturn value * factor;\n} catch (error) {\n console.error('Error converting units:', error);\n return value;\n}\n}\n\nexport default ValidationHelpers;
```

## 1.5 memory-manager.js

javascript

```

/**
 * Memory Manager for CVD Risk Toolkit
 *
 * Manages memory usage, prevents memory leaks, and
 * implements resource cleanup for optimal performance.
 */

class MemoryManager {
 static #cachedData = new Map();
 static #maxCacheItems = 20;
 static #maxCacheAge = 30 * 60 * 1000; // 30 minutes

 /**
 * Initialize memory management
 */
 static initialize() {
 console.log('Initializing memory management...');

 // Set up periodic cleanup
 setInterval(this.cleanupCache.bind(this), 5 * 60 * 1000); // Every 5 minutes

 // Set up memory pressure handler for modern browsers
 if ('gc' in window) {
 window.addEventListener('memorypressure', this.handleMemoryPressure.bind(this));
 }

 // Initialize ResultManager for pagination
 this.resultManager = new ResultManager();
 }

 /**
 * Cache data with expiration
 * @param {string} key - Cache key
 * @param {*} data - Data to cache
 * @param {Object} options - Options
 */
 static cacheData(key, data, {
 expiration = this.#maxCacheAge,
 priority = 0
 } = {}) {
 // Enforce cache size limit
 if (this.#cachedData.size >= this.#maxCacheItems) {
 this.cleanupCache();
 }
 this.#cachedData.set(key, {
 data,
 expiration,
 priority
 });
 }
}

```

```

 // If still at limit, remove oldest item
 if (this.#cachedData.size >= this.#maxCacheItems) {
 let oldestKey = null;
 let oldestTime = Infinity;

 for (const [k, v] of this.#cachedData.entries()) {
 if (v.timestamp < oldestTime && v.priority <= priority) {
 oldestTime = v.timestamp;
 oldestKey = k;
 }
 }

 if (oldestKey) {
 this.#cachedData.delete(oldestKey);
 }
 }
}

// Add to cache with timestamp
this.#cachedData.set(key, {
 data,
 timestamp: Date.now(),
 expiration,
 priority
});
}

/**
 * Get cached data
 * @param {string} key - Cache key
 * @returns {*} - Cached data or null if not found or expired
 */
static getCachedData(key) {
 const cached = this.#cachedData.get(key);

 if (!cached) {
 return null;
 }

 // Check if expired
 if (Date.now() - cached.timestamp > cached.expiration) {
 this.#cachedData.delete(key);
 return null;
 }

 return cached;
}

```

```

 }

 return cached.data;
}

/***
 * Clear entire cache or specific key
 * @param {string} key - Optional key to clear
 */
static clearCache(key = null) {
 if (key) {
 this.#cachedData.delete(key);
 } else {
 this.#cachedData.clear();
 }
}

/***
 * Cleanup expired cache items
 */
static cleanupCache() {
 const now = Date.now();

 for (const [key, value] of this.#cachedData.entries()) {
 if (now - value.timestamp > value.expiration) {
 this.#cachedData.delete(key);
 }
 }
}

/***
 * Handle memory pressure events
 * @param {Event} event - Memory pressure event
 */
static handleMemoryPressure(event) {
 console.warn(`Memory pressure detected: ${event.pressure}`);

 // Clear non-critical cache on high pressure
 if (event.pressure === 'critical') {
 this.#cachedData.clear();
 this.resultManager.clearOldResults();

 // Force garbage collection if available
 if (typeof window.gc === 'function') {

```

```

 window.gc();
 }
} else {
 // Clean up expired items
 this.cleanupCache();
}
}

/**
 * Add calculation result to manager
 * @param {*} result - Calculation result
 * @returns {*} - The result
 */
static addResult(result) {
 return this.resultManager.addResult(result);
}

/**
 * Get paginated results
 * @param {number} page - Page number (0-based)
 * @param {number} resultsPerPage - Results per page
 * @returns {Array} - Results for the page
 */
static getResultsForPage(page = 0, resultsPerPage = 5) {
 return this.resultManager.getResultsForPage(page, resultsPerPage);
}

}

/**
 * Result Manager Class for paginating and managing results
 */
class ResultManager {
 constructor(maxStoredResults = 10) {
 this.results = [];
 this.maxStoredResults = maxStoredResults;
 this.currentPage = 0;
 }

}

/**
 * Add a result to the manager
 * @param {*} result - Result to add
 * @returns {*} - The result
 */
addResult(result) {

```

```

 // Add to beginning of array
 this.results.unshift(result);

 // Trim if exceeding maximum
 if (this.results.length > this.maxStoredResults) {
 this.results = this.results.slice(0, this.maxStoredResults);
 }

 // Reset to first page
 this.currentPage = 0;

 return this.results[0];
}

/**
 * Get results for a specific page
 * @param {number} page - Page number (0-based)
 * @param {number} resultsPerPage - Results per page
 * @returns {Array} - Results for the page
 */
getResultsForPage(page = 0, resultsPerPage = 5) {
 const start = page * resultsPerPage;
 const end = start + resultsPerPage;
 return this.results.slice(start, end);
}

/**
 * Clear all but the most recent result
 */
clearOldResults() {
 if (this.results.length > 1) {
 this.results = [this.results[0]];
 this.currentPage = 0;
 }
}

export default MemoryManager;
```
``` factors for common unit types
const factors = {

```