**Implementation Plan for Enhanced QuadrantAssignmentContext**

**Current Situation**

The application currently uses a QuadrantAssignmentContext that:

* Stores midpoint information
* Tracks manual quadrant assignments
* Provides a basic getQuadrantForPoint function
* Calculates distribution statistics

The main limitation is that terminology (classic vs. modern) isn't centrally managed, leading to inconsistencies in how data points are labeled across different components. Each component has to independently apply terminology rules.

**Target Implementation (Option 4)**

Enhance the existing context with additional classification functions that centralize terminology handling while maintaining flexibility for different display requirements.

**Files to Modify**

1. **Primary File**: src/components/visualization/context/QuadrantAssignmentContext.tsx
   * This will require the most significant changes to enhance the context functionality
2. **Component Files**:
   * src/components/visualization/components/DataPoints/DataPointInfoBox.tsx
   * src/components/visualization/components/DataPoints/DataPointRenderer.tsx
   * src/components/reporting/components/DistributionSection/DistributionSection.tsx
   * Any other components that display quadrant labels

**Required Changes**

**1. Enhance QuadrantAssignmentContext.tsx**

typescript

*// Add to the interface:*

interface QuadrantAssignmentContextType {

*// Existing properties...*

*// Add terminology management*

isClassicModel: boolean;

setIsClassicModel: (isClassic: boolean) => void;

*// Add enhanced classification functions*

getDisplayNameForQuadrant: (quadrantType: QuadrantType) => string;

isPointInSpecialZone: (point: DataPoint) => boolean;

getHierarchicalClassification: (

point: DataPoint

) => {

baseQuadrant: QuadrantType;

specificZone: QuadrantType | null;

};

*// Existing methods...*

}

*// Update provider implementation*

export const QuadrantAssignmentProvider: React.FC<QuadrantAssignmentProviderProps> = ({

children,

data,

satisfactionScale,

loyaltyScale,

initialMidpoint,

isClassicModel = true, *// Add this prop*

}) => {

*// Existing state...*

const [isClassicModelState, setIsClassicModelState] = useState(isClassicModel);

*// Implement new functions*

const getDisplayNameForQuadrant = (quadrantType: QuadrantType): string => {

if (!isClassicModelState) {

if (quadrantType === 'apostles') return 'Advocates';

if (quadrantType === 'terrorists') return 'Trolls';

}

*// Capitalize first letter*

return quadrantType.charAt(0).toUpperCase() + quadrantType.slice(1);

};

const isPointInSpecialZone = (point: DataPoint): boolean => {

const quadrant = getQuadrantForPoint(point);

return ['apostles', 'terrorists', 'near\_apostles', 'near\_terrorists'].includes(quadrant);

};

const getHierarchicalClassification = (

point: DataPoint

): {

baseQuadrant: QuadrantType;

specificZone: QuadrantType | null;

} => {

const quadrant = getQuadrantForPoint(point);

*// Map special zones to their base quadrant*

let baseQuadrant: QuadrantType = quadrant;

let specificZone: QuadrantType | null = null;

if (quadrant === 'apostles' || quadrant === 'near\_apostles') {

baseQuadrant = 'loyalists';

specificZone = quadrant;

} else if (quadrant === 'terrorists' || quadrant === 'near\_terrorists') {

baseQuadrant = 'defectors';

specificZone = quadrant;

}

return { baseQuadrant, specificZone };

};

*// Update context value*

const contextValue: QuadrantAssignmentContextType = {

*// Existing values...*

isClassicModel: isClassicModelState,

setIsClassicModel: setIsClassicModelState,

getDisplayNameForQuadrant,

isPointInSpecialZone,

getHierarchicalClassification,

};

return (

<QuadrantAssignmentContext.Provider value={contextValue}>

{children}

</QuadrantAssignmentContext.Provider>

);

};

**2. Update DataPointInfoBox.tsx**

typescript

export const InfoBox: React.FC<{

point: DataPoint;

normalized: { x: number; y: number };

quadrantInfo: QuadrantOption;

count: number;

samePoints: DataPoint[];

availableOptions?: QuadrantOption[];

onGroupChange?: (group: QuadrantOption) => void;

}> = ({ point, normalized, quadrantInfo, count, samePoints, availableOptions, onGroupChange }) => {

*// Get context*

const { getDisplayNameForQuadrant, getQuadrantForPoint } = useQuadrantAssignment();

*// Replace direct group name with context function*

const displayGroupName = getDisplayNameForQuadrant(getQuadrantForPoint(point) as QuadrantType);

*// Rest of component...*

*// Replace in render:*

<div className="data-point-info\_\_value" style={{ color: quadrantInfo.color, fontWeight: 600 }}>

{displayGroupName}

</div>

}

**3. Update DataPointRenderer.tsx**

Update to use the new context functions.

**4. Update DistributionSection.tsx**

Update to use the new context functions.

**Impact Analysis**

**Positive Impacts**

* **Consistent terminology**: All components will display the same terms for the same data points
* **Simplified components**: Components no longer need their own terminology conversion logic
* **Centralized control**: Changes to terminology behavior only need to be made in one place
* **Hierarchical awareness**: The context now understands the relationship between quadrants and special zones

**Potential Challenges**

* **Backward compatibility**: Existing components expecting direct quadrant strings may need updates
* **Performance**: New context functions add slight computational overhead, but this should be minimal
* **Component props**: Some components may need to be updated to remove redundant props if they now get this info from context

**Implementation Steps**

1. Update QuadrantAssignmentContext.tsx with the new functions and properties
2. Update App.tsx to pass the isClassicModel prop to the QuadrantAssignmentProvider
3. Update DataPointInfoBox.tsx to use the context functions
4. Update DataPointRenderer.tsx to use the context functions
5. Update any other components that display quadrant information
6. Test thoroughly to ensure consistent display throughout the application

**Future Considerations**

* **Filter context**: Consider whether to extend this context to handle filters or create a separate filter context
* **More granular classification**: As the application grows, the context might need to provide more specialized classification functions
* **Caching**: If performance becomes an issue, consider caching results of classification functions
* **Context splitting**: If the context grows too large, consider splitting into separate contexts for classification, filters, etc.

This plan provides a roadmap for implementing Option 4 while minimizing disruption to the existing architecture.