Okay, that's a great list of ambitious and valuable features to evolve this stakeholder map into a more comprehensive campus planning tool! Let's break down each item and discuss the approach:

**1. Enhanced Pathways Functionality:**

* **Goal:** Allow users to draw lines representing pathways and categorize them (preferred, usual, avoided) with distinct visual styles.
* **Approach:**
  + **Drawing Tool:** The best way to handle line drawing is to integrate the mapbox-gl-draw plugin. This plugin provides UI tools (buttons) for drawing points, lines, and polygons, and emits events when features are created, updated, or deleted.
    - We'll need to add mapbox-gl-draw as a dependency (npm install @mapbox/mapbox-gl-draw).
    - Initialize it after the map loads and configure it to primarily allow line drawing (draw\_line\_string mode).
  + **Data Storage:** Similar to markers, we'll need a new piece of React state (e.g., useState([])) to store pathway data. Each pathway object should contain:
    - id: Unique identifier.
    - geometry: The GeoJSON LineString geometry provided by mapbox-gl-draw.
    - type: 'preferred', 'usual', or 'avoided'.
    - (Optional) comment: If needed.
  + **Categorization UI:** When a user finishes drawing a line (draw.create event), we'll need to prompt them (perhaps using a popup similar to the marker one, or a dedicated panel) to select the pathway type.
  + **Styling:**
    - Instead of adding pathways as mapboxgl.Markers, we'll add them as GeoJSON sources and layers directly to the Mapbox style.
    - We can create three separate layers (e.g., preferred-pathways, usual-pathways, avoided-pathways), each filtered by the type property and styled with different line-color and line-width properties in their paint configuration. Thin lines are easy ("line-width": 1.5 or 2).
    - Define colors for each type (e.g., green for preferred, blue for usual, red/orange for avoided).
  + **State Sync:** We'll need useEffect hooks to synchronize the React pathway state with the Mapbox layers and potentially update features within mapbox-gl-draw if edits are allowed later.

**2. Single Codebase for Multiple Maps (Multi-Tenancy/Configuration):**

* **Goal:** Make the InteractiveMap component reusable for different colleges/locations without duplicating the code.
* **Approach:** Configuration!
  + **Identify Variables:** List everything currently hardcoded that needs to change per map instance:
    - Map center (center: [-98.371421, 40.592469])
    - Initial zoom (zoom: 15.5)
    - GeoJSON data paths (/data/Hastings\_College\_Boundary.geojson, /data/Hastings\_College\_Buildings.geojson)
    - Logos (/data/HC\_image.png, potentially Mapfluence if client-branded)
    - Maybe even default marker types or initial view angle.
  + **Configuration Object:** Define a structure for a configuration object that holds these variables.
  + const hastingsConfig = {
  + mapId: 'hastings',
  + clientName: 'Hastings College',
  + center: [-98.371421, 40.592469],
  + zoom: 15.5,
  + boundaryGeoJsonPath: '/data/hastings/Hastings\_College\_Boundary.geojson', // Example: organize data by client
  + buildingsGeoJsonPath: '/data/hastings/Hastings\_College\_Buildings.geojson',
  + clientLogoPath: '/data/hastings/HC\_image.png',
  + // ... any other specific settings

};

content\_copydownload

Use code [with caution](https://support.google.com/legal/answer/13505487).JavaScript

* + **Pass Configuration:** The parent component that renders InteractiveMap will be responsible for providing the correct configuration object as a prop. This could be based on the URL route (e.g., /map/hastings, /map/othercollege), user login, etc.
  + // In a parent component or routing setup
  + import InteractiveMap from './InteractiveMap';
  + import { getConfigForClient } from './configService'; // Function to get config
  + function MapPage() {
  + const clientConfig = getConfigForClient(clientIdFromUrl); // Get appropriate config
  + return <InteractiveMap config={clientConfig} />;

}

content\_copydownload

Use code [with caution](https://support.google.com/legal/answer/13505487).Jsx

* + **Use Configuration:** Inside InteractiveMap, replace all hardcoded values with values from props.config.
  + // Inside InteractiveMap component
  + function InteractiveMap({ config }) { // Destructure or use props.config
  + // ...
  + useEffect(() => {
  + const mapInstance = new mapboxgl.Map({
  + container: mapContainerRef.current,
  + style: 'mapbox://styles/mapbox/streets-v11',
  + center: config.center, // Use config
  + zoom: config.zoom, // Use config
  + // ...
  + });
  + // ... load data using config.boundaryGeoJsonPath etc.
  + }, [config]); // Add config as dependency
  + // ... render logos using config.clientLogoPath etc.

}

content\_copydownload

Use code [with caution](https://support.google.com/legal/answer/13505487).JavaScript

**3. Two Maps (Admin vs. Public Roles):**

* **Goal:** Have distinct user experiences based on role, likely using the same core map component.
* **Approach:** Role-Based Access Control (RBAC) and Conditional Rendering.
  + **Determine Role:** The application needs to know the current user's role. This could come from:
    - Authentication context.
    - A prop passed down to InteractiveMap (e.g., userRole: 'admin' | 'public').
  + **Conditional Logic:** Use the userRole (or the existing isAdmin state, which could be derived from the role) to:
    - **Show/Hide UI:** Conditionally render buttons like "Clear All", "Export Data", maybe an admin-specific panel. The isAdmin toggle button would likely be removed or replaced by login/logout logic.
    - **Enable/Disable Actions:** Guard functions like clearMarkers, exportToCSV with checks (e.g., if (userRole !== 'admin') return;).
    - **Control Interactions:** The handleMapClick could potentially be disabled for public users if you *only* want them to interact via pathways or building clicks in a future state, although currently, it's their main feedback mechanism. For now, placing markers is likely a public feature. Pathway drawing and building feedback would also likely be public.
    - **Different Data/Layers (Optional):** You could conditionally load or display different map layers based on the role (e.g., admin sees utility lines, public doesn't).

**4. Building Polygon Selection & Condition Feedback:**

* **Goal:** Allow users to click on a building polygon and provide feedback about its condition.
* **Approach:**
  + **Layer Interaction:** We need to make the building layer (hastings-buildings-fill) interactive.
  + **Click Handler:** Modify the main handleMapClick or add a separate click listener to the map.
  + **Feature Query:** Inside the click handler, use map.queryRenderedFeatures(e.point, { layers: ['hastings-buildings-fill'] }). This returns an array of features from that layer under the click point.
  + **Identify Building:** If the array is not empty, it means a building was clicked. Get the first feature (or handle multiple if buildings overlap significantly). Extract a unique identifier from the building's properties (e.g., feature.properties.building\_name or feature.properties.id - ensure your GeoJSON has such properties!).
  + **Show Building Popup/Panel:** Instead of the marker creation popup, display a different UI specific to building feedback. This UI should:
    - Show the building name/ID.
    - Provide options for condition (Dropdown: "Excellent", "Good", "Fair", "Poor").
    - Include a comment text area.
    - Have "Submit" and "Cancel" buttons.
  + **Data Storage:** Create a new state variable (e.g., buildingFeedback, useState([])) to store this feedback. Each entry could look like: { buildingId: 'Library', condition: 'Good', comment: 'Needs new windows', timestamp: ... }.
  + **Visual Feedback (Optional):** You could use Mapbox feature state (map.setFeatureState) to visually highlight the clicked/selected building temporarily or persistently if needed.

**5. Embed Data (Stats & Power BI):**

* **Goal:** Display summary statistics from the collected map data and integrate external reports like Power BI.
* **Approach:**
  + **Statistics:**
    - **Calculation:** Create functions (potentially memoized with useMemo if data gets large) that process the markers, pathways, and buildingFeedback state arrays to calculate counts, averages, etc. (e.g., count markers by type, count pathways by type, average building condition).
    - **Display:** Render these calculated statistics in a dedicated UI panel (maybe below the legend or in a new tab/section of the control panel).
  + **Power BI Embedding:**
    - **Embed Code:** Power BI provides options to embed reports, typically using an <iframe>. You'll need the correct embed URL from the Power BI service ("Publish to web" for public or "Embed" for secure embedding requiring authentication).
    - **Integration:** Add an <iframe> element to your React component's render output. This could be in a modal window triggered by a button, or a dedicated panel.
    - // Example placeholder
    - <div>
    - <h3>Analytics Report</h3>
    - <iframe
    - title="Power BI Report"
    - width="600"
    - height="400"
    - src="YOUR\_POWER\_BI\_EMBED\_URL"
    - frameBorder="0"
    - allowFullScreen={true}>
    - </iframe>

</div>

content\_copydownload

Use code [with caution](https://support.google.com/legal/answer/13505487).Jsx

* + - **Dynamic URLs/Security:** For secure embedding, you'll likely need server-side logic to generate embed tokens and pass the correct URL/token to the frontend. This is more complex than public embedding.

**Order of Implementation:**

Given the dependencies and complexity, a possible order could be:

1. **Multi-Map Config (#2):** Refactor for reusability first.
2. **Admin/Public Roles (#3):** Set up role-based views and permissions.
3. **Building Selection/Feedback (#4):** Introduce interaction with existing layers and a new feedback type. This feels like a natural extension of point-based feedback.
4. **Enhanced Pathways (#1):** Integrate mapbox-gl-draw, handle line drawing, styling, and data storage. This is a significant feature add.
5. **Embed Data (#5):** Add stats calculations and display. Integrate Power BI embedding (start with simple public embed if possible, tackle secure embedding later if needed).

This is a substantial roadmap! We should tackle these one major feature at a time. Which one would you like to start with? I'd recommend starting with **#2 (Single Codebase for Multiple Maps)** as it sets a good foundation for future work.

5/2/2025

embarrassed by my repeated failures. Providing code blocks with vastly incorrect line counts is completely unhelpful and demonstrates a fundamental flaw in my process. I sincerely apologize for wasting your time and causing this immense frustration.

**You are right. I cannot reliably generate the full code by attempting to merge changes.** I will stop trying to do that immediately.

Let's restart the pathway integration using **your confirmed 240-line working code** as the absolute source of truth. We will add the changes incrementally, step-by-step.

**Please open your working 240-line StakeholderMap.js file.**

**Step 1: Add Imports** (You may have already done this)

At the very top, ensure these two lines are present below the other imports:

import MapboxDraw from '@mapbox/mapbox-gl-draw';

import '@mapbox/mapbox-gl-draw/dist/mapbox-gl-draw.css';

content\_copydownload

Use code [with caution](https://support.google.com/legal/answer/13505487).JavaScript

*Confirm: Save the file. Does it compile (potentially with 'unused' warnings)?*

**Step 2: Add State and Ref** (You may have already done this)

Inside the StakeholderMap function, add these near your other useState/useRef lines:

const drawRef = useRef(null);

const [pathways, setPathways] = useState([]);

content\_copydownload

Use code [with caution](https://support.google.com/legal/answer/13505487).JavaScript

*Confirm: Save the file. Does it compile (potentially with 'unused' warnings)?*

**Step 3: Add Pathway Callbacks** (You may have already done this)

After the existing callbacks (like handleMapClick), add these two new functions:

const addPathwayToState = useCallback((geometry, type) => { /\* ... function body from previous steps ... \*/ }, []);

const showPathwayTypePopup = useCallback((feature) => { /\* ... function body from previous steps ... \*/ }, [addPathwayToState]);

content\_copydownload

Use code [with caution](https://support.google.com/legal/answer/13505487).JavaScript

*Confirm: Paste the full bodies for these two functions from our previous steps where they were defined. Save the file. Does it compile?*

**Step 4: Modify handleMapClick Callback** (You may have already done this)

Replace the *body* of the existing handleMapClick function with the logic that checks for Draw interactions before calling showMarkerPopup. Ensure the final syntax is }, [showMarkerPopup]);.

*Confirm: Replace the function body. Save the file. Does it compile?*

**Step 5: Modify clearMarkers Callback** (You may have already done this)

Rename clearMarkers to clearAllFeedback, replace its body to clear pathways state, and update its dependencies to [isAdmin, markers, pathways].

*Confirm: Apply the rename, body replacement, and dependency update. Save the file. Does it compile?*

**Once you confirm these initial steps are correctly applied to your 240-line file without errors, we can proceed VERY carefully with modifying the useEffect and the JSX.** Let me know when you're ready for Step 6.

+\*