ChatGPT

Executive Summary

The goal is to **rewrite the Modular Membrane UI** around the 3D donut canvas with a fresh, modular panel system. The redesign introduces a flexible shell with three display modes (Hidden, Overlay, Docked) and two innovative panel access modes ("Circle" and "Desk"). We will implement a **vanilla HTML/CSS/JS** architecture that cleanly separates the Membrane Directory (shell navigation) from individual content panels. Key improvements include a **radial "Circle Mode" menu** (small circular chips arrayed around the donut for quick access) and a **bottom "Desk Mode" bar** (a strip of icons for minimized or grouped panels) for handling larger numbers of panels. Each panel will feature a consistent header with title, a status indicator, and control icons (pin, focus, collapse, close) for intuitive window management. We'll also enhance the **UX polish** with smooth scrolling, inset custom scrollbars, and glassmorphic styling (using transparency and backdrop blur). Keyboard shortcuts (e.g. **Cmd/Ctrl+K** to open search, **Esc** to close panels) and intelligent auto-hiding of UI elements (the shell hide button disappears when nothing is open) round out the improved experience.

In summary, this plan provides a **detailed, implementation-ready blueprint** for the new Membrane panel system. It covers a modular component breakdown, state management with persistence, UX specifications for the shell directory and panels, algorithms for circle and desk layouts, step-by-step migration from the current codebase, and ideas for subtle enhancements. All decisions are grounded in the existing code structure and the redesign requirements, ensuring the final implementation will integrate smoothly with the 3D donut environment and provide an engaging, futuristic interface for users.

Architecture Overview (Components & Modules)

The rewritten system will be organized into clear UI components and modules, each handling a specific aspect of the interface. Below is a breakdown of the major components and their relationships:

- **Shell (Membrane Directory Panel):** The shell is an `<aside>` element that contains the Membrane directory (list of all membrane panels) and controls for showing/hiding or switching modes. It uses the existing `#sidebarShell` container with classes for modes (`is-hidden`, `is-overlay`, `is-docked`) [1] [2]. This shell can be toggled or repositioned but remains the anchor for navigation. Inside the shell, the directory list is a scrollable list of membrane entries (with search, filter, sort UI above it).

- **Floating Layer (Panel Windows Layer):** A container (`<div id="floatingLayer">`) that holds all individual panel `<section>` elements as floating windows [3]. Panels are initially hidden (via `hidden` attribute or `.is-hidden` class) and are shown when opened. They remain children of this layer when in "floating" (free window) mode. The floating layer covers the main 3D canvas, and panels within it have `position: relative; z-index: 10; pointer-events: auto` so they sit above the canvas [4]. We will maintain this layer for all active panel content.

- **Radial Menu (Circle Mode UI):** A new overlay element (e.g. `<div id="menuRing" class="menu-ring">`) that will hold **circular chip buttons** positioned around the donut. Each chip corresponds to a membrane panel (likely the pinned or frequently used ones). The `menuRing` can be a full-screen fixed container (similar to `floatingDockLayer`) with pointer-events none on background and auto on chips [5]. Within it, we'll dynamically position child elements (chips) in a circle around the canvas center. This component will have its own module (e.g. `radialMenu.js`) responsible for computing positions and handling interactions (hover, click). The radial menu is shown when Circle Mode is active and the shell is hidden.

- **Bottom Dock (Desk Mode UI):** A new **desk bar** element (e.g. `<div id="deskBar" class="desk-bar">`) that sits fixed at the bottom of the screen. It will contain a horizontal list of **chip icons** (similar to a taskbar) representing panels that have been minimized or bundled to the desktop. The desk bar can be implemented inside the existing `#floatingDockLayer` (which is already a fixed full-viewport overlay [6]) or as a sibling element. We will likely create a wrapper inside `floatingDockLayer` for the desk bar to allow clipping and scrolling. This component's module will handle adding/removing chips when panels are collapsed to the bar, and restoring panels when clicked.

- **Panel Windows (Membrane Panels):** Each content panel remains a `<section class="panel membrane-panel">` element (as in the current HTML structure). We will enforce a consistent internal structure for these panels:

- A **header** (e.g. `<header class="panel__header">`) containing the panel title, a status dot, and control buttons (pin, focus, collapse, close).
- A **body/content** area (which holds the panel's unique controls, forms, visualizations as currently implemented).
- Panels have identifying attributes: `id="somePanelId"`, `data-membrane-name` (display name), and new attributes if needed (like `data-membrane-category` for filtering). Many existing panels already have `data-membrane-name` and some have special classes like `membrane-panel--circular` or `--desktop` which we'll repurpose [7].
- Panel positioning: when floating, panels can be dragged and will maintain their `style.top/left` (we'll use an inline style or CSS transform for position). When in circle or desk mode, the panel itself is hidden and represented by a chip elsewhere (the panel element might be moved out of `floatingLayer` or simply kept hidden in place).

- Panel modules: We will write or refactor a **Panel Manager module** to handle opening/closing panels, applying the correct mode (float, circle, desk) to each panel, and updating state.

- **State Management Module:** We will extend the existing `state` (imported from `state.js`) to include UI state for the panel system [8]. This will include:

- `state.ui.shellMode` – current shell display mode (`'hidden'`, `'overlay'`, `'docked'`).
- `state.ui.pinnedPanels` – an array or set of panel IDs that are pinned as favorites (max 5).
- `state.ui.circleMode` / `state.ui.deskMode` – booleans or a single enum to indicate whether the radial menu or desk bar is currently active for representing minimized panels.
- `state.ui.openPanels` – list of panel IDs currently open (for persistence of open state, if desired).

- `state.ui.recentPanels` – list of recently opened panel IDs (to support "Recently Opened" sorting).
- `state.ui.filters` – object with filter toggles for each category (Signals/Geometry/Visual/Meta) in the directory.
- `state.ui.sort` – current sort mode (e.g. `'alpha'` for A–Z, `'recent'`, or `'pinned'`).
- These state fields will be saved via the existing `persist()` function to localStorage [9] so that user preferences (like pinned items or last used mode) persist across sessions.

- **Event & Interaction Handlers:** We will attach event listeners for user interactions:

- Shell toggle button clicks (to cycle or set Hidden/Overlay/Docked mode).
- Search input events (to update filter in real-time).
- Filter and sort control events (to re-render the directory list).
- Panel header button clicks:
  - Pin button -> toggles that panel's pinned status.
  - Focus button -> brings that panel to front (if open) or opens+focuses it.
  - Collapse button -> minimizes the panel to a chip (either radial or desk, depending on mode).
  - Close button -> closes the panel (hides it).
- Radial chip hover -> triggers expansion (show label) for that chip.
- Radial chip click -> toggles the corresponding panel open or closed.
- Desk bar chip click -> restores the panel window (or toggles it).

- Keyboard events:

  - `Ctrl/Cmd + K` -> focus the search bar (and show the shell overlay if it was hidden).
  - `Esc` -> if a panel is focused/open, close it (or if the directory overlay is open with no panel focused, close/hide the directory).

- **Styling and Themes:** We continue using the `light.css` (and analogous dark if any) with new styles:

- CSS variables (e.g. `--panel-bg`, `--panel-border`, `--glass`) will be used for the translucent panel backgrounds and borders. The dev journal indicates panel backgrounds are now transparent by default [10], so we will leverage `backdrop-filter: blur(...)` on panel backgrounds to create a frosted-glass effect.
- **Inset scrollbars:** We'll customize scrollbars inside the shell and panels. The CSS already sets a stable scrollbar gutter in `.sidebar` [11]. We can refine this or add a thin overlay scrollbar track that appears on hover. Each panel's content area (`overflow-y: auto`) will get similar treatment using WebKit scrollbar CSS (and a fallback for Firefox using scrollbar-width thin).
- **Animations:** We'll add smooth transitions for panel movements and chip hovers. For example, using `transition: transform 0.2s, opacity 0.2s` on chips to animate expansion, and CSS keyframes for subtle entrance of panels.
- **Responsiveness:** While focusing on desktop, we'll structure CSS in a way that can extend to touch (larger hit areas on chips, alternative to hover). For instance, chips on touch could always display labels or use a long-press to "hover."

The architecture separates concerns: the **Shell Directory module** manages listing and searching membranes; the **Panel Manager** controls panel state and movement; the **Radial Menu** and **Desk Bar** provide alternate UIs for minimized panels. This modularity will make the code easier to maintain and extend, and it aligns with the existing code structure (which already has separate modules like `initCirclesPanel`, etc.). Diagrammatically:

- **UI Containers:** (shell aside) ↔ (floatingLayer) ↔ (dockLayer for circle/desk)
- **State & Logic:** (state.ui & persist) ⟷ (shell controls & panel controls event handlers) ⟷ (render functions for directory, radial, desk)
- **Panels:** each panel listens to state or events (e.g. a Brain Wave panel might update its status dot via events when EEG connects).

By clearly delineating these components, we ensure a **cohesive yet modular system**. Next, we'll detail each part of the specification: state management, shell directory UX, panel UX, circle mode, desk mode, migration plan, and extras.

State & Persistence Specification

We will extend the app's state schema to support the panel system, leveraging the existing `state` object and `persist()` mechanism for local storage. All new state fields will live under a logical namespace (likely `state.ui` or `state.membrane`), as the current code uses `state.ui` for UI flags (e.g. `state.ui.entryDoor`, `state.ui.scenePreset` etc.). Key state and persistence points:

- **Shell Mode (`state.ui.shellMode`):** Stores the current shell display mode: `'hidden'`, `'overlay'`, or `'docked'`. Toggling the shell (via the sidebar toggle button) will update this state and persist it. On app load, we can apply the last used mode (falling back to `'overlay'` as a default if not set). This ties into the existing `setSidebarMode()` which adds the appropriate class to `#sidebarShell` [12] and updates the toggle button label [13]. We will ensure `state.ui.shellMode` stays in sync with those class changes. Example: `state.ui.shellMode = 'overlay'` means we add `.is-overlay` class and remove others on `#sidebarShell`, and we'd persist that so next load the shell comes up overlay by default.

- **Pinned Panels (`state.ui.pinnedPanels`):** An array (or object set) of panel IDs that are pinned as favorites. This is limited to 3–5 entries. Pinning a panel (via its header pin button or via the directory list) will add it to this array (if below limit) and persist. Unpinning removes it. The pinned list influences:

- The **directory list** rendering: pinned panels are shown in a "Pinned" section at the top of the list for quick access.
- The **Circle mode chips**: We plan for Circle Mode to primarily display pinned items (since those are the high-priority panels the user wants quick access to). Thus, `state.ui.pinnedPanels` drives which chips appear around the donut.
- We will enforce the max count: if the user tries to pin beyond the limit, the UI could either refuse (disable the pin action or show a notice) or automatically unpin the oldest entry. We'll likely simply disable further pin action once 5 are pinned – e.g., the pin button becomes disabled or hidden if at

max. (This decision can be refined with user input, but we'll implement the safe route of disallowing >5 pinned).

- Persistence: Pinned list will be saved so that favorites persist on reload.

- **Open Panels (`state.ui.openPanels` or deriving from DOM):** We want to remember which panels are currently open (and possibly their last position on screen) so that on refresh or re-entry, the workspace can be restored. We can maintain an array of open panel IDs. Each time a panel is opened or closed, update this array. We will also store each panel's **mode/state** if needed:

- Perhaps an object mapping panel ID -> `{ mode: 'floating'|'circular'|'desktop', x, y }`. The code already has `panelDockState` for this purpose [14] [15], which tracks mode and slot ("left/right/floating") and coordinates. We will integrate with that. For example, if a panel is floated, we store its `x,y` position; if it's collapsed to circle or desk, we store that mode. This map is persisted via `persist()` as part of `state` (assuming we put it under `state.ui.dockStates`).
- However, since our new design drops explicit left/right docking for panels, our `mode` values simplify to: `'floating'` (open window), `'circle'` (minimized to radial chip), `'desk'` (minimized to bottom bar), or `'closed'` (not in openPanels list at all).
- We will likely reuse the `panelDockState` structure from current code but adapt it. Currently, the code uses `panelDockState` and marks mode/slot and `membrane-panel--circular/--desktop` classes [7]. We can leverage that approach:
    - When a panel is toggled to circle mode, mark `panelDockState[id].mode = 'circle'` and add class `membrane-panel--circular` to the element (and conversely remove if not).
    - For desk mode, `mode = 'desktop'` and class `membrane-panel--desktop`.
    - Floating (default open) can be `'floating'` with no special class (or remove those mode classes).
    - Hidden/closed panels are simply not in `openPanels`, and their `hidden` attribute (or CSS) is set.

- On load, after creating DOM, we iterate over saved `state.ui.openPanels` and reopen those panels in the appropriate mode. For each saved panel in circle/desktop mode, we won't auto-open their content, but we will immediately create their chip in the radial menu or desk bar (so that the user sees their last session's minimized panels ready to activate). If a panel was left floating open, we will open it (remove `hidden`, append to floatingLayer) and restore its last x,y coordinates from state (ensuring it appears where the user left it).

- **Search and Filter State:**

- We'll store the current search query `state.ui.searchQuery` (string). This might not need persistence (could start empty on each session), but it will be maintained in state while the UI is open for debouncing or other logic. The current code uses a local `membraneSearchTerm` variable [16]. We can tie it to state to simplify passing it around, but it's not critical to persist.
- Filter toggles: `state.ui.filterSignals/Geometry/Visual/Meta` (booleans) or one object `state.ui.filters = { signals: true, geometry: true, visual: true, meta: true }`. These represent which categories are currently included in the directory listing. If the user changes these (via some UI control like checkboxes or pills), update state and re-render the list. We

can persist filter choices if it makes sense (so the directory remembers "I only want to see Visual panels" for example on next load).

- Sort mode: `state.ui.sortMode`, with allowed values `'alpha'` (alphabetical A–Z by name), `'recent'` (recently opened first), `'pinned'` (pinned first – though pinned are anyway separate section). Default can be `'alpha'`. Changing sort updates state and triggers re-sort of list. Persist this so that user's preferred sort is kept.

- **Panel Metadata & Status:**

- We will introduce a static mapping or data attribute for each panel's **category** and perhaps an icon. The easiest is to add `data-membrane-category="Signals"` (or "Geometry"/"Visual"/"Meta") in the HTML for each panel section. This way, the directory render function can read `panel.dataset.membraneCategory` to filter. If adding to HTML is not desirable, we can maintain a JS map of panelId->category.
- We will also maintain a mapping of panelId -> display name (though `data-membrane-name` already serves as the display label for most panels [17]).
- Panel **status** (for the status dot indicator): This might not be a single state variable since it depends on panel-specific logic (e.g., brainwave connection status, etc.). Instead, we'll update a CSS class or data attribute on the panel element when its internal state changes. For example, if the Neurosity Crown panel connects to a device, our Crown code can do `document.getElementById('crownPanel').dataset.status = 'online'` and we style the dot accordingly (green). We will define a few generic statuses: `'default'` (or no status, meaning inactive/idle), `'online'` (active/running, green dot), `'error'` (if any panel had an error state, red dot), `'offline'` (explicit offline state, maybe gray or orange). Each panel's own logic can set this state.

- Persisting panel status isn't necessary beyond the session – it reflects runtime conditions. So we won't store it in localStorage (except if needed for restoration of UI state, but e.g. if something was running, on reload it likely resets anyway).

- **Persistence Implementation:** We will integrate with the existing `persist()` function [8]. Likely `persist()` already serializes `state` to `localStorage` (the presence of `state` and calls to persist after changes in the code suggests this is in place [18]). We must ensure all our new `state.ui` properties are serializable (no DOM nodes, just primitives/arrays). When reading state on load (likely via something like `defaultFrames` or some init function), we'll merge persisted state with defaults.

- For example, define default values:

```
state.ui = {
  shellMode: 'overlay',
  pinnedPanels: [],
  openPanels: [],
  filters: { signals:true, geometry:true, visual:true, meta:true },
  sortMode: 'alpha'
```

```
  // ...
};
```

Then load persisted overrides (if any) and apply.

- Because this is a **rewrite/major update**, we might also consider versioning the state. If the old version state had different structure (like `panelDockState` etc.), we should either migrate it or reset aspects. To avoid confusion, we can store our new UI state under a distinct key (maybe `state.uiVersion = 2` or such) or simply ensure backwards compatibility by reading `panelDockState` if exists and translating it. For instance, if `panelDockState` from old state has some panels pinned left/right, we interpret that as open (but since ghost sidebars removed, all those would become floating or closed). It might be acceptable to drop old pinning on the first run of the new system (since it was experimental).

- **Example**: If a user last had `shellMode = 'docked'`, `pinnedPanels = ['brainwavePanel','crownPanel']`, and had `unitCirclePanel` open and minimized to desk, the state might look like:

```
"ui": {
  "shellMode": "docked",
  "pinnedPanels": ["brainwavePanel","crownPanel"],
  "openPanels": ["unitCirclePanel"],
  "dockStates": {
     "unitCirclePanel": { "mode": "desktop" }
  },
  "filters": { "signals": true, "geometry": true, "visual": true, "meta":
true },
  "sortMode": "alpha"
}
```

On load, we attach `#sidebarShell.is-docked` and show the directory, render pinned items on top of list (Brain Waves & Crown), and place a chip for Unit Circle in the desk bar (since it was in desktop mode). The user immediately sees Crown/BrainWave chips around the donut (Circle mode for pinned) and a Unit Circle icon in the bottom bar. Everything else is closed. The state then evolves as user interacts, and each significant change calls `persist()` to save.

In summary, the state spec ensures that **user customizations and UI context persist** seamlessly. Favorites, last open panels, and UI mode preferences will not be lost. We leverage the robust persistence pattern already in the code (state + persist) to implement this. By carefully updating `state.ui` on each user action (toggling shell, pinning, opening, etc.), we maintain a single source of truth for the UI, which simplifies rendering logic (the render functions can refer to state instead of DOM queries wherever possible). This state-driven approach also makes it easier to add future features like preset workspaces or undo/redo of layout changes.

Shell/Directory UX Specification

The shell (Membrane Directory) is the user's primary navigation hub for all membrane panels. It will be redesigned as a **transparent, minimal panel** that can overlay or dock, containing search, filtering, sorting, and a list of panels (with sections for pinned and others). Below are the UX details:

- **Shell Modes (Hidden, Overlay, Docked):** The shell lives in the `#sidebarShell` aside element and toggles between:
- **Hidden:** Not visible at all (CSS `display: none`). In this mode, the user instead relies on Circle or Desk chips for access. The shell toggle button itself might remain visible as a small "menu" button if needed to reopen the directory. (If the design calls for an always-available way to open the directory when hidden, we could repurpose the current sidebar toggle button to show as a small icon when shell is hidden.)
- **Overlay:** The shell appears as an overlay panel on top of the canvas, likely anchored to the right side of the screen (as per current CSS `.sidebar-shell.is-overlay { display: flex; right: 16px; }` [1]). In overlay, the donut canvas is still fully visible behind it (shell has a translucent background). This is ideal for quick access without permanently consuming space.
- **Docked:** The shell is fixed to a side (we'll use left side as default). Current CSS `.sidebar-shell.is-docked` moves it to left:16px [1] . In Docked mode, the directory is intended to feel like a side panel that's part of the layout. It will typically be taller (stretch top to bottom) and push the main content slightly (though since our canvas is full-screen behind, "pushing" just means it overlaps less of it on the left). We ensure the docked shell doesn't obscure critical parts of the donut UI (the donut is centered, so left docking is okay).
- **Transition:** Users toggle modes via the shell toggle button (text changes "Overlay"/"Docked"/"Hidden"). We maintain the cycling logic (Hidden -> Overlay -> Docked -> Hidden). The implementation uses `cycleSidebarMode()` as in current code [19] , updated to reflect any label changes ("Hidden" might be renamed "Hide" etc.). We'll also allow direct setting: e.g., if the user drags/resizes the shell to dock it, but such direct manipulation is not currently in scope, so cycle button suffices.

- The shell's **toggle button** is the existing `#sidebarShellToggle` button in the header [20] . We will keep its basic structure, but possibly adjust styling (maybe an icon instead of text, to be more minimal). We will implement logic so that **when the shell has no content, the toggle (hide) button auto-hides**. "No content" in this context means no panels listed – which could theoretically happen if all panels are removed (unlikely), or more realistically if the directory is empty because filters exclude everything. In practice, since there will always be at least some panel available, this auto-hide may refer to hiding the toggle when shell is not in use (e.g., shell hidden mode). We interpret the requirement such that if shell is hidden and nothing is "active" in it, we don't even show the on-screen toggle button. Instead, perhaps the presence of any open or pinned panel could cause a small "menu" button to appear for the user to reopen the shell. Conversely, if the user has no panels open and shell is empty, even that button could fade out until the user triggers it via a keyboard shortcut or a specific gesture. We will implement a simple rule: if `state.ui.shellMode === 'hidden'` and `state.ui.pinnedPanels.length === 0` and no search query is active, hide the toggle UI entirely (maybe except a tiny stub if needed). Once a panel gets pinned or opened, we can fade the toggle back in. This addresses the "auto-hides when shell is empty" requirement gracefully.

- **Header of Shell (Title & Controls):** The shell's header (as defined in HTML) contains a title "Membrane" and the toggle button [20]. We will keep the title text (or possibly change to an icon logo if provided). To the right of the title, the toggle button switches modes as described. We might add an explicit close "⨯" button as well if we want to allow hiding directly (though the cycle button covers that by cycling to Hidden). For clarity, we might instead change the toggle button to a single-function "hide/show" (like a collapse arrow) and have separate controls in UI to switch overlay vs dock. However, given the spec lists 3 modes, it implies a single control cycling is acceptable. We will label it clearly (perhaps use icons: an open eye for overlay, a pinned icon for dock, and a hidden eye for hide, with tooltip labels).

- The shell header's appearance: we'll use a subtle translucent background (using CSS variable `--panel-bg` if defined, which is currently `transparent` [21]; we might override it with something like `rgba(0,0,0,0.2)` plus `backdrop-filter: blur(5px)` for a glass look). The header text and button use the accent or light text color.

- **Search Bar:** Immediately below the shell header, we place the search input. In HTML, we have:

```
<div class="membrane-search membrane-search--directory">
  <label for="membraneSearchInput">Search membranes</label>
  <input type="search" id="membraneSearchInput"
placeholder="Type to find a sub-brane">
</div>
```

as seen in the existing code [22]. We will preserve this structure but enhance it:

- The search input (`#membraneSearchInput`) will have **focus autofocus** when the user presses **Cmd/Ctrl+K** globally. We'll add a global keydown listener to trigger this:

```
document.addEventListener('keydown', (e) => {
  if((e.metaKey || e.ctrlKey) && e.key.toLowerCase() === 'k') {
    e.preventDefault();
    openShell('overlay'); // ensure shell visible
    membraneSearchInput.focus();
    membraneSearchInput.select();
  }
});
```

This provides the quick panel search similar to command palette behavior.

- As the user types, we filter the list in real-time. The current implementation `renderMembraneDirectory()` already filters by name [23] [24]. We will refine `matchesMembraneLabel()` to also consider partial matches and maybe keywords. The search will likely match against panel name and possibly tags (we could have a hidden data attribute for tags like synonyms).

- We should also offer a clear "×" button on the right side of the search field to clear the query (for convenience). This can be an `<button type="button" class="search-clear">×</button>` absolutely positioned within the input field wrapper, shown when `value.length > 0`.

- Accessibility: ensure the label is properly associated (it is, via for/id). The aria-live region of the list is already polite in the code [25], meaning updates announce the number of results.

- **Filter Controls:** We will include filter toggles for categories **Signals, Geometry, Visual, Meta**. UX-wise, these could be small pill-style toggle buttons in a row below the search bar, or a dropdown. Given space in the aside, we can show them as four toggle buttons (with possibly icon + label or just short label):

- Example:

```
<div class="membrane-filters">
  <button type="button" class="filter-btn is-active" data-
filter="signals">Signals</button>
  <button type="button" class="filter-btn is-active" data-
filter="geometry">Geometry</button>
  <button type="button" class="filter-btn is-active" data-
filter="visual">Visual</button>
  <button type="button" class="filter-btn is-active" data-
filter="meta">Meta</button>
</div>
```

  Initially all filters "on" (is-active). Clicking one toggles it off (gray out the button). The directory list should update to show only panels whose category matches an active filter. For instance, if the user toggles off "Visual", any panel with `data-membrane-category="Visual"` will be excluded from the list.
- Implementation: when a filter button is toggled, we update `state.ui.filters[category]` and re-run the render of the list. The filtering logic will be inserted into `renderMembraneDirectory()`: we already gather all panels and filter out fixed ones [26]; we will add:

```
.filter(panel => {
  const cat = panel.dataset.membraneCategory || 'Meta';
  return state.ui.filters[cat.toLowerCase()];
});
```

  so only allowed categories are listed.
- If all filters are off (unlikely we allow that; maybe ensure at least one stays on), we could either show nothing with a message or automatically re-enable one. Perhaps simply allow it and show "No membranes available" (which already happens if none pass the filter, thanks to the empty-state logic [27]).
- The filter categories mapping: We will assign each panel a category:
    - **Signals:** e.g. *Neurosity Crown*, *Brain Waves*, any EEG/OSC or data signal panels.

    - **Geometry:** e.g. *Donuscope Builder* (torus geometry settings), *Unit Circle* panels (if they relate to geometry of circles), *Platonic Perspective*, *Alignment*—anything configuring shapes or layout.
    - **Visual:** e.g. *Everything Chalice* (visual stacked donuts), *Field Command Deck*, *Language of Light*, any panel controlling rendering, colors, glimmer, etc.
    - **Meta:** e.g. *Entry Door*, *Membrane Directory* (though directory panel itself won't be listed), *Intentions + Personal Labels* (since that's more about user meta info), or any conceptual/high-level panels. We will refine these assignments with the team, but the code will be ready for those categories.

- The filter UI should be intuitive: when none of a category's panels are present, maybe the button could disable, but since categories cover all, that's not needed. We will include a brief tooltip on hover of filter buttons (e.g. "Show/hide Signal category panels") for clarity.

- **Sort Controls:** The user can sort the non-pinned portion of the list by different criteria:

- **A–Z:** Alphabetical by panel name.
- **Recently Opened:** Based on a timestamp of last open (we track this per panel in state on open, and perhaps store in `state.ui.recentPanels` or a separate map `state.ui.lastOpenedTimes`). This lets frequently used or recently viewed items rise to top.
- **(Pinned first):** Pinned items are anyway separated at top, so this likely refers to within the main list if we choose to also intermix pinned vs others when sort is "pinned". But since we explicitly will show pinned in their own section at top always, we may not need a "pinned" sort option. Instead, pinned zone is always on top by design.
- Another possible sort is by category or custom grouping, but not requested explicitly.
- Implementation: A simple approach is a drop-down select:

```
<select id="membraneSortSelect">
  <option value="alpha">A–Z</option>
  <option value="recent">Recently Opened</option>
</select>
```

  or we could do toggle buttons for each sort mode (but only one active at a time, so radio behavior). A drop-down might be cleaner in the UI header (less clutter).
- When sort changes, update `state.ui.sortMode` and re-render the list. The `panels` array in `renderMembraneDirectory` can be sorted accordingly:

```
if(state.ui.sortMode === 'alpha'){
   panels.sort((a,b)=> aName.localeCompare(bName));
} else if(state.ui.sortMode === 'recent'){
   panels.sort((a,b)=> {
      const at = state.ui.lastOpened?.[a.id] || 0;
      const bt = state.ui.lastOpened?.[b.id] || 0;
      return bt - at;
   });
}
```

We maintain `state.ui.lastOpened` as a map of panelId -> timestamp (updated in openPanel function). If none present, treat as 0 so they fall back.

- We will ensure pinned panels are *excluded* from this sort operation and always rendered first in their own pinned section. The code will actually generate two sections:
    1. **Pinned Section:** If `state.ui.pinnedPanels` is non-empty, create a fragment for pinned items. Iterate pinned panel IDs in whatever order (we could decide to sort pinned alphabetically or in pin order – likely in the order the user pinned them, or allow manual reordering, but that's out of scope now, so pin order = the array order).
    2. **All others:** Filter out panels that are pinned, then apply search/filter, then sort as selected. We then concatenate: if pinned section exists, maybe prepend a small heading or separator line "Pinned:" then list those, then maybe another separator "All Membranes:" then the sorted list.
- The UI for sections can be simple: e.g. a `<div class="membrane-directory__section-title">Pinned</div>` at top of pinned block for clarity (styled subtly). If no pinned, no title needed.

- If sort mode is changed to "recent", we won't move the pinned section; it remains on top. Only the lower section reorders. This provides consistency (pinned are always easily reachable).

- **Directory List Entries:** Each panel is listed as a row with its name and status, plus actions:

- The current HTML generated (from `renderMembraneDirectory()` in code) creates:

```
<div class="membrane-directory__item" data-membrane-id="panelId">
  <div class="membrane-directory__meta">
    <span class="membrane-directory__name">Panel Name</span>
    <span class="membrane-directory__status">StatusText</span>
  </div>
  <div class="membrane-directory__actions"> ... buttons ... </div>
</div>
```

as we see in code [28] [29] . We will adjust this:

    - The **name** remains as the clickable or focusable text of the item (we might allow clicking the name to open the panel too, not just the action buttons – making the whole item clickable to open might be user-friendly). Perhaps clicking the row (outside of specific buttons) will open (or focus) the panel.
    - The **status** text currently shows things like "Docked Left · Hidden" etc. [30] . We will simplify that. Instead of text, we'll use a **status dot icon**:
    - We add an element, e.g. `<span class="status-dot status-dot--green"></span>` (with classes like `--green`, `--red`, etc. based on panel status). This dot will visually convey status (e.g. green = active, gray = idle, etc.). We'll place it likely before or after the name. Possibly at the very left of the item for quick scanning.
    - For accessibility, the dot can have `aria-hidden="true"` and we keep a text in the status span for screen readers like "Online" or "Offline". Alternatively, we use the existing status span but instead of showing "Hidden/Docked", repurpose it to show status text (like "● Online" with a colored dot via CSS).

- For simplicity: use `.membrane-directory__status` span to hold a • character or an SVG circle, colored via CSS, followed by a short status word. E.g., `<span class="membrane-directory__status is-online">●</span>` with CSS coloring and `::after` to add tooltip “online”. But since specific statuses vary by panel, we might just update its text dynamically (like current code does with baseStatus and statusParts [31]).
- We will remove the old “Docked/Hidden” from statusParts since that concept is changing (panels won’t be docked in sidebars, and hidden status isn’t needed because if it’s closed it simply won’t be in openPanels list).
- Instead, statusParts could include things like “Pinned” or “Open” if we want to denote those. But likely not needed: pinned is visually indicated by maybe a pin icon in actions, and open panels we could highlight differently (maybe list item bold or an “open” icon). However, since the user can see open panels on screen, listing might not need an explicit label.
- **Actions:** We will update the action buttons for each list item:
- Remove “Pin L”, “Pin R”, “Float” textual buttons [32] (those were for the old docking logic) [33]. Instead, provide at most 2 actions in the list:
    1. A **Pin/Unpin** toggle (to favorite the panel). Represented by a pin icon (e.g. or a font-awesome pin if available). If the panel is already pinned, show a filled pin and clicking will unpin (remove from favorites). If not pinned, show an outlined pin and clicking pins it (adds to favorites). We disable it or hide if pinned count is maxed and this panel isn’t pinned.
    2. An **Open/Focus** button. We can combine open & focus into one action: basically a “launch” button (triangle play icon or arrow). If the panel is closed, clicking it will **open** the panel (in floating mode by default). If the panel is already open but perhaps behind others, clicking could just bring it to front (focus it). Either way, it ensures the panel’s content is visible. We could use an icon like or ⇢, but perhaps a simple “Open” text or an eye icon (to “view” the panel). Since the entire row could be clickable to open as well, we might not need a separate open button – but to be safe for accessibility (e.g. using keyboard, you can tab to the open button), we’ll include it.
    3. Alternatively, we use double-click or Enter on the list item to open, and use the focus button exclusively to bring to front if needed. But double-click might not be obvious, so a button is clearer.
- We might omit a direct “Close” here because closing from the list doesn’t make sense if it’s not open. If it is open, we could change the open button to “Close” instead, but better to close from the panel’s own UI or maybe via right-click context if we add that. To keep it simple, we won’t add a close in the directory list (the user can close via panel header or Esc key).
- These will be implemented as icon buttons with appropriate classes, e.g.:

```
actionMeta = [];
if(panelId in state.ui.pinnedPanels) {
  actionMeta.push({action: 'unpin', label: 'Unpin'});
} else {
  actionMeta.push({action: 'pin', label: 'Pin', disabled:
state.ui.pinnedPanels.length >= 5});
}
actionMeta.push({action: 'open', label: 'Open'}); // label can be
'Focus' if open already
```

and we create buttons for each. The current code structure for building actions can be reused [32]. We'll map these actions in an event handler:

```
membraneDirectoryList.addEventListener('click', (e)=>{
  const btn = e.target.closest('[data-membrane-action]');
  if(!btn) return;
  const action = btn.dataset.membraneAction;
  const panelId = btn.dataset.membraneId;
  switch(action){
    case 'pin': pinPanel(panelId); break;
    case 'unpin': unpinPanel(panelId); break;
    case 'open': openPanel(panelId); break;
    // etc.
  }
});
```

(The code currently has a similar event delegation for actions [34].)

- Visual for actions: using icons will reduce clutter. We can style `.ghost-btn` (used currently) or define new classes for these icon buttons (maybe `.icon-btn`). They'll likely appear on hover or always visible to the right of each row. For example, a pin icon and an "open" icon (which could be an arrow or eye). We should also consider keyboard accessibility: each item should be reachable and actionable via keyboard. Possibly, the list items can be made focusable (role="button" or anchor) so Enter opens them, and the pin can be tabbed to as a button. We'll ensure proper ARIA labels on the icon buttons (e.g. `aria-label="Pin panel"`).

- **Pinned Section in List:** Pinned panels appear at the top under a heading "Pinned" (if any pinned exist). Each pinned item's entry may not need the pin button (since it's redundant, they're pinned already) or we show it as "Unpin". We could choose to not show the pin action for pinned items at all (just an indicator they're pinned), but giving the option to unpin is useful. So likely we keep the unpin button on them. They otherwise look like any list entry. We might visually emphasize pinned items (e.g. a ☆ icon or a slight highlight background) to denote their importance. This is an aesthetic choice – maybe a subtle highlight or a pin icon right next to their name.

- We enforce max 5 pinned, so this section will never overflow too large. The directory can comfortably show up to 5 pinned plus others.

- **Empty States:**

- If search + filters yield no results, we display a message. The code already handles this by adding a `<p class="membrane-directory__empty">No membranes match your search.</p>` [35]. We will update the messaging to also account for filters (e.g. if filters exclude everything, say "No panels in this category"). Possibly combine: "No membranes found. Adjust your search or filters." for any case of empty.

- If for some reason the entire directory has no panels (which in a properly configured app shouldn't happen), we might show "No membranes available." (already in code for empty case with no search [36]).

- We will also hide the "Hide shell" button in this case as mentioned, because an empty shell is not useful.

- **Shell Scroll & Size:** The shell (aside) will scroll internally if content is long. We ensure a max-height (already in CSS: `.sidebar-shell .sidebar-shell__body { flex:1; overflow-y:auto; max-height: calc(100vh - 40px); }` [37]). We will test that pinned + all items fit or scroll as needed. The scrollbars will be styled to be minimal (as discussed in styling). We can implement **smooth scrolling**: if programmatically scrolling (e.g. focusing a newly added element), use `behavior: smooth`. The user's own scroll via mousewheel is as usual.

- **Keyboard Navigation:**

- When the shell directory is focused (we can allow the container to be focusable or at least the search input is initial focus), pressing **Tab** should cycle through filter controls, sort control, then into the list of items. Each directory item can receive focus – perhaps we wrap the content in an `<a href="#">` to naturally focusable, or give the `.membrane-directory__item` a `tabindex="0"` and outline style on focus. Pressing **Enter** when an item is focused will open that panel (same as clicking it). We'll implement this by capturing keydown on the item element when focused.
- Arrow keys: It would be nice to allow using up/down arrows to move between items. We can give the container `role="listbox"` and items `role="option"` for ARIA semantics, and implement arrow key handling (on keydown in container: if arrow down, focus the next `.membrane-directory__item`).
- Esc key: If the shell is open (overlay or docked) and user presses Esc, we will hide the shell (go to hidden mode) *only if* no panel is focused. We have to decide priority: pressing Esc when directory is open could close the directory *instead* of closing an open panel. We can implement that if directory has focus or search input has focus, Esc closes the directory; if a panel has focus (or generally if any panel modal is open and maybe the directory not in use), Esc closes that panel. We will differentiate by context:
    - Add an event on search input and directory that on Esc, sets `shellMode='hidden'`.
    - Panels themselves may handle Esc if they have modals, but in general we can use a global Esc handler: if shell overlay visible, hide it, else if any panel open, close the top one. This might conflict if user intends to close panel first, but typically if directory is open it means user is interacting with it. We will implement a reasonable rule and document it.

Overall, the Shell/Directory will function as a **command center** for the user: it's where they discover panels (via search and categories) and manage favorites. It will feel lighter and more transparent than before (no heavy background). The user can keep it open docked for a "control panel" vibe, or hide it and just use the immersive radial/desk controls. By providing both quick-access (circle/desk) and a comprehensive directory (shell), we cover novice and power user needs. All these behaviors are grounded in the current design (e.g., the existing code's structure for listing and toggling panels [38] [29]) but greatly refined to meet the new spec.

Panel UX Specification (Window Behavior & Controls)

Each membrane panel (the content sections like *Neurosity Crown*, *Brain Waves*, *Field Deck*, etc.) will be presented as a **floating panel window** with a unified look and set of controls. This section details how individual panels behave, how they can be moved/resized, and the function of the header controls (pin, focus, close, collapse). We ensure these panels integrate smoothly with the Circle/Desk modes and the shell.

- **Appearance & Layout:** Panels will have a consistent styling:
- **Size:** Most panels have a predefined size (via their content). We will allow panels to have a **minimum width** (say 300px) and possibly a maximum width (maybe 500-600px for very large ones), and they can scroll internally if content overflows vertically. We won't implement free resizing in this iteration (could be a future enhancement), except maybe some panels like builder might already respond to parent width.
- **Position:** When opened, a panel will by default appear centered in the viewport (or slightly offset if multiple opened to avoid exact overlap). The code currently computes a `stageCenterX/Y` for panel placement [39]. We will use similar logic: open new panel at center (or next to last opened panel by some offset to avoid stacking exactly).
- Panels are **draggable**: We will implement click-and-drag on the panel header (and possibly an empty area) to move them. This likely was partially supported (the code's `initDockablePanel` sets up placeholders and such for dragging/docking [40] [41]). We can simplify: use the HTML5 Drag API or pointer events. We'll attach `onmousedown` on `.panel__header` that starts tracking pointer and moves the element (setting `position: absolute` or adjusting transform). Because panels are inside `#floatingLayer` which is fixed, we can position them with CSS `top` / `left`. We'll update those as the user drags, and add `pointer-events: none` to iframe or canvas inside if any (to avoid capturing events, if panels have interactive graphics). Many panels are just controls, so no conflict.
- We snap panel position to grid if desired (there is mention of `MEMBRANE_GRID_SIZE = 200` in code [42] maybe for alignment), but we likely allow free positioning. We *will* constrain movement so the panel doesn't drag completely off-screen (at least part of it stays visible).
- **Depth/Focus:** Only one panel can be "focused" (top) at a time. When a panel is clicked (anywhere inside), we bring it to front:
    - We will use either CSS z-index or DOM reorder for stacking. The current `.membrane-panel` all have the same z-index by CSS [4], so last in DOM wins. We can manage focus by moving the focused panel's element to the end of `floatingLayer` child list, or by applying an inline style `z-index: 11` (and increment for subsequent focuses to avoid infinite build-up, maybe reset occasionally). We will implement a helper `bringToFront(panelEl)` that increments a global z-counter and assigns it to panelEl.style.zIndex, also track currently focused panel in state (for keyboard focusing, etc.).
    - Focus state might also add a class `.is-focused` to highlight the border or header of that panel visually (maybe glow or accent border).
- No panel is modal (except maybe Entry Door which is special). Users can have multiple open and switch between them.

- **Header & Title:** We will standardize a header bar at the top of each panel:

- Contained in a `<header class="panel__header">` (some existing panels use a `<header>` or a `<h2>` directly). We will modify HTML of each panel to ensure:

```
<header class="panel__header">
  <h2 class="panel__title">Panel Name</h2>
  <div class="panel__controls">
     <!-- buttons will go here -->
  </div>
</header>
```

  If a panel already has a `<header>` with content, we'll integrate our controls into it or replace it with this standardized version (preserving any subtitle or meta info as needed). For instance, *Brain Waves* panel has a complex header with mode buttons and a collapse button [43] [44]. We may keep those domain-specific controls but include our common controls on the right side of the same header bar.
- The **Panel Title** (`.panel__title`) displays `data-membrane-name` or a short title. We ensure a consistent font and size (perhaps reuse the styles from existing `.panel__title`, which might be defined globally or at least many panels use it).
- The header might also show a small status dot indicator for that panel's status (particularly if the panel deals with live data). We can include an element `<span class="panel__status-dot"></span>` to the left of the title text. E.g. in Brain Waves header, there is already a status light element (class `.brainwave-panel__light`) [45]. We will generalize this:
  - The dot can be a small SVG or CSS circle, color controlled by a class or data attribute (like `.is-online` = green, `.is-offline` = gray, `.is-error` = red).
  - We will update this dot in real-time based on panel state changes (for Brain Waves, when connected, turn green, etc.). The panel's specific code can call a generic function or just toggle the class on its element.
  - The dot has a tooltip or ARIA label for status ("● Online" etc.) if needed.
- **Controls in Header:** We include four controls as specified:
  1. **Pin (Favorite):** Icon of a pin. Clicking it pins or unpins the panel. This is effectively a duplicate of the directory's pin action, but provided in context for convenience. If the panel is not currently pinned as favorite, clicking pin will add it to `state.ui.pinnedPanels` (and thus show it in radial menu, etc.). If already pinned, clicking will unpin. We reflect the icon state (filled vs outlined) accordingly. This control is separate from "window pinning" (not to be confused with docking). It's purely marking favorite. We will place this as a toggle button that highlights when active. Also, if pinned count is max and this panel isn't pinned, we might disable the button (or better, allow it by unpinning the oldest automatically – but that could confuse, so we'll probably disable with a tooltip "Unpin another panel first").
  2. Class: `.panel-btn--pin`.
  3. Title/aria: "Pin this panel to favorites" (or "Unpin…" when active).
  4. Behavior: `onclick -> togglePin(panelId)`.
  5. **Focus:** Icon of a spotlight/eyeball/bring-to-front. If multiple panels are open, clicking this one will simply call the bringToFront function on this panel, raising its z-index above others and maybe giving it a little attention animation (like a quick border pulse to indicate focus gained). If the panel was minimized (circle/desk), focus could also mean "open and focus" – but in that case the panel would not have a visible header with controls (because it's minimized). So this button is mainly relevant when panel is open. It's somewhat optional

because clicking anywhere on a panel will focus it too. However, having a dedicated focus button can be useful in the directory list (as earlier) or if we consider a scenario with keyboard: user navigates to a background panel's header and hits the focus button to ensure it's front.

6. Class: `.panel-btn--focus`.
7. Title: "Focus (bring to front)" – only needed if multiple open; if this panel is already top or only one, clicking focus might do nothing visible (we could decide to maximize the panel to a default position/size if we had that concept, but we are not implementing maximize beyond bring front).
8. We may grey it out or hide it if only one panel open or if none other behind it.
9. **Collapse:** Icon of a minimize/underline or an arrow-down into a box. Clicking collapse will **minimize the panel** into either a radial chip or the desk bar, depending on which mode is active/preferred:
10. If the environment is in Circle Mode (i.e., user primarily using radial chips and shell is hidden), collapsing should convert the panel into a **circle chip around the donut**. That means:
    - Add the class `membrane-panel--circular` to the panel element (marking its state) [7].
    - Hide the panel's content (maybe just apply `hidden` attribute or remove it from DOM into a placeholder as the code did with `placeholder` node [46]).
    - Add a new chip to the radial menu for this panel if not already there. If the panel was not pinned, we'll treat this as a dynamic chip (the panel is currently shelved to circle).
    - In state, set `panelDockState[id].mode = 'circle'` (or update our `state.ui.openPanels` mode mapping).
    - Visually, the panel disappears from the screen and the user sees a chip icon in the circle. (We might animate this transition: e.g., panel shrinks towards its chip location).
11. If in Desk Mode context (e.g., a lot of panels or user prefers bottom bar), collapsing will **bundle the panel to the bottom bar**:
    - Add class `membrane-panel--desktop` to the panel element.
    - Move or hide the panel element similarly, and create an icon in the desk bar for it.
    - Update state mode = 'desktop'.
12. If neither mode is explicitly active, we decide based on number of open panels: possibly default to desk if many. Alternatively, we always collapse to desk bar unless explicitly user toggled a "circle mode view." We might use a simple heuristic: if panel is pinned (i.e., has a radial chip already) then collapse to radial (since there is a natural place for it), otherwise collapse to desk bar. This way pinned favorites minimize around donut, others go to bottom bar.
13. The **collapse button** thus covers the function of the code's "bundle" (desk) and "circle" toggle buttons that were in the old control panel UI [47] [48]. We simplify to one button that decides target automatically, but we will ensure the user can achieve either mode if desired (possibly via a global toggle or settings which we address in Circle/Desk mode spec). For now, collapse does what's appropriate.
14. Class: `.panel-btn--collapse`.
15. Title: "Minimize panel" (maybe dynamically "Minimize to ring" vs "Minimize to dock" depending on context).
16. **Close:** Icon of an "X". Clicking closes the panel completely:
17. Remove it from `openPanels` state, hide its element (`hidden` attribute, add `.is-hidden` class).

18. If it was minimized, remove its chip too.
19. If it was pinned, we do **not** unpin on close (pin is a separate concept). The user might close a pinned panel – it remains in favorites for next time, just not currently open.
20. If panel had unsaved data or something, we might prompt (not likely necessary in this app).
21. Class: `.panel-btn--close`.
22. Title: "Close panel".
23. After closing, if no panels remain open and shell is hidden, we might show the shell or at least the donut is fully clear. If shell toggle was hidden due to empty, it might fade in now to prompt user they can open menu – but since pinned chips could still be there, it's fine.

- **Header Control Layout:** We'll arrange these buttons in the top-right corner of the panel header. They will likely be small (16x16 or 20x20px icons). We can either show them only on hover of the panel (like window controls often show on hover/title bar) or always show. Given the aesthetic, always showing faint icons that become bright on hover is fine. We will maintain a consistent order: Pin, Focus, Collapse, Close. We could also group pin & focus to left side of header and collapse/close to right side, but it's easier to group all right-aligned. Possibly we'll right-align all four in a row.
    - We will add CSS such as:

```
.panel__controls { display: flex; gap: 8px; }
.panel__controls .panel-btn { background: transparent; border: none;
color: #aaa; }
.panel__controls .panel-btn:hover { color: #fff; }
```

    Each button might have an icon via content or using an `<svg>` inside.

- For panels that already had their own header controls (like Brain Waves had collapse and mode toggles, etc.), we'll integrate carefully. For example, Brain Waves mode buttons (Raw/Bands/Waves) could stay in header to left side, and our controls to right side. The collapse button in brainwave (which collapsed its graph section) might be redundant with our collapse (which collapses whole panel). We might remove the brainwave-specific one to avoid confusion (likely we will, since the new collapse covers it globally).

- **Panel Content & Behavior:**

- The content area of panels (below header) remains as is from the code: various controls (sliders, toggles, etc.). We will ensure that when a panel is collapsed or closed, any **active processes** are handled:
    - For instance, if Brain Waves panel is streaming EEG and the user closes it, do we stop the stream or keep it running in background? Possibly stop, as user closed it. We can call the appropriate teardown (like `crownAdapter.disconnect()` or so). Similarly, if collapsed to chip, do we pause updates or keep running? Perhaps keep running if it's meant to continue (like music in background). That's product-level decision. We might assume panels control something persistently, and closing just hides UI but does not disable the feature, unless explicitly "off" toggled.
    - However, given complexity, a safe approach is: closing a panel = turning off its feature (unless it's pinned meaning might still want it? Unlikely, pin is just favorite).
    - We'll coordinate with each feature dev: e.g. *Neurosity Crown* panel – maybe it should disconnect if panel closed to save resources. If user reopens, they can reconnect.

- Scrolling inside panels: Panels with lengthy content (like lists or lots of sliders) have their own scrollbar. We'll style these to be *inset*, meaning the scrollbar appears within panel boundary (not the entire viewport). Already, panels like `.sidebar` had scrollbar styling [11]. We can apply similar styles to `.membrane-panel` if needed: a thin track that becomes visible on hover.

- Some panels (like *Everything Chalice*) might have visuals that could overflow panel boundaries. We ensure `overflow: auto` on panel body, so content is clipped to panel area (no bleed outside).

- **Inter-panel interactions:**

- If multiple panels are open, they behave like independent windows. But we might implement slight **collision avoidance**: e.g., when a new panel opens, if it would cover an existing one completely, we offset it. Or if the user drags panels such that they overlap, that's allowed. We won't implement complex collision or tiling in this iteration (could be an enhancement).
- The user can manually arrange them. We trust them to do so or use collapse to manage space.

- We will, however, implement that the **active panel's z-index is top** and maybe drop shadows to differentiate. Already `.membrane-panel` likely has a shadow (we see `--shadow` in CSS variables [49]). We can intensify the shadow for focused panel for depth effect.

- **Animation & Micro-Interactions:**

- Opening a panel: we can add a brief fade-in or slide-in. Perhaps scale from 0.9 to 1 and fade.
- Focusing (bringing front): a quick subtle flash or border glow to draw attention.
- Collapsing: animate panel shrinking into its icon (we can do this by animating the panel's width/height down and opacity, while simultaneously animating the chip from small to normal size – but a simpler approximation is fine given time).

- These are enhancements; the core is functionality, but we will include these if not too complex.

- **Accessibility & ARIA:**

- Each panel section should have `aria-label` or an accessible name (the title serves as one). We might add `role="dialog"` to panels when open to signify pop-up windows, with `aria-labelledby` pointing to the title element.
- The header controls all get `aria-label` attributes as mentioned for screen readers.
- Keyboard: when a panel opens, we might want to auto-focus something in it (maybe the first input, or just the panel container so that pressing Tab will go into its fields). This can be handled by focusing the first form control or the panel itself on open.
- Pressing **Esc** while a panel has focus closes that panel (common behavior for popups). We'll implement that: add keydown on document that if `Esc` and a panel is topmost/focused, call closePanel on it. We must ensure not to override if an input inside needs Esc (rare). We can check `event.target.tagName` to not do it if in a text input maybe.

To illustrate, consider the *Brain Waves* panel after implementing this spec: - It will have a header bar with title "Brain Waves" and a green status dot if EEG streaming. On the right, four icons: pin (if user clicks, it pins Brain Waves to favorites), focus (if multiple open, brings Brain Waves front), collapse (minimize to either a donut chip or bottom icon), close (X out the panel). The panel can be dragged around. The user can collapse

it to a chip, which then might show as an EEG icon at the bottom or around donut, blinking to indicate still streaming (if we keep it running). They can click that chip to restore the panel. If they close it, the EEG stream stops and the panel is gone until reopened via directory (or chip if pinned). - The *Neurosity Crown* panel similarly can be pinned (maybe user pins it because they often use it), collapsed when they don't need the form open (maybe it then still runs in background indicated by the dot on its pinned chip). - Panels like *Field Command Deck* with many sliders can be left open while adjusting, or collapsed to get it out of the way and quickly reopened from dock.

This unified panel UX ensures every panel is manageable in a consistent way, reducing cognitive load. The header controls in particular give the user immediate options without having to go back to the directory for those actions. We have aligned these features with the existing code signals (the code's `initDockablePanel` was injecting similar controls [50] [51], including circle and desk toggles which we incorporate, and an off button which maps to our Close). Thus, we're largely implementing what was planned, but simplifying the UI and tying it to explicit user actions rather than hidden gestures.

Circle Mode Specification & Algorithm (Radial Menu)

Circle Mode provides an immersive, around-the-donut interface for accessing panels via radial "chips." In this mode, small circular icons (chips) are arranged in a ring around the donut's canvas, allowing quick hover and click interactions. This section specifies how Circle Mode works and details the algorithm for laying out chips.

- **When Circle Mode is Active:** Circle mode is primarily used when the shell is hidden or when the user wants minimal UI. We anticipate two uses:
- **Pinned-Favorites Access:** By default, we'll show **pinned panel chips** around the donut. This means even if the shell (directory) is closed, the user can still open their favorite panels via these chips.
- **Minimized Panel Indication:** If the user collapses an open panel and we choose to represent it in the circle (per our collapse logic), those collapsed panels also appear as chips (even if not pinned).
- Essentially, any panel in state with mode `'circle'` will have a chip in the radial menu.

- The radial menu can coexist with desk bar, but typically if few items we'll use radial.

- **Radial Menu Structure:** We have a container (e.g. `<div id="menuRing" class="menu-ring">`) that is full-screen but pointer-events none except its child chip elements [52]. Within it:

```
<div class="menu-ring__slots">
  <!-- dynamically filled -->
  <button class="menu-ring__slot" data-panel-id="brainwavePanel"
title="Brain Waves">
    <span class="menu-ring__icon"> </span>
    <span class="menu-ring__label">Brain Waves</span>
  </button>
  ... (other slots)
</div>
```

We'll create one `menu-ring__slot` for each chip. We use a `<button>` so it's focusable and clickable. It contains:

- An icon element (could be an `<img>` or emoji or font icon) for quick recognition.
- A label span with the panel name (hidden by default via CSS, shown on hover). We might not wrap them in additional circle elements because we can directly position the buttons in a circle using CSS transforms. Alternatively, we could absolutely position each chip with inline styles.

- **Chip Iconography:** We need an icon for each panel:

- If available, we could use a relevant emoji or icon. For example:
  - Brain Waves: (wave emoji) or a waveform icon.
  - Crown (EEG): or some neural icon.
  - Everything Chalice: or (donut) for cosmic donut?
  - Field Deck: 🎛 (control knobs).
  - etc.
- If an icon set is not readily available, we can use the first letter of the panel name as a pseudo-icon (perhaps styled as a circle with that letter, similar to a monogram). The code's existing UI might not have icons for all; so a letter or short code could suffice (e.g., "BW" for Brain Waves).
- We will define either in a config or as data attributes on panels (like `data-icon=" "`) to fetch the appropriate symbol.

- **Layout Algorithm:**

- Let `N` = number of chips to display in circle. The algorithm distributes them evenly on an imaginary circle.
- Determine the center of the donut canvas in screen coordinates. If the canvas is full screen and donut is centered, the center is roughly at `(window.innerWidth/2, window.innerHeight/2)`. We might refine by any offsets (if UI elements offset canvas).
- Choose a radius for placement: likely just outside the donut. If the donut has a known radius in pixels (maybe via THREE.js we can derive approximate projection size), or simpler, use a fraction of window size. For example, `radius = min(window.innerWidth, window.innerHeight) * 0.33` (one-third of screen) might place chips around the donut nicely. We can also make radius dynamic if donut zooms, but donut likely static scale in screen.
- Compute positions for each chip i:
  - Angle step = 360° / N.
  - We need a starting angle offset. Often for aesthetic, start at top (90°) or slightly off top center so chips don't collide with toggle UI if at top. Let's start at -90° (which is top if 0° is east). Then angle_i = -90° + i * (360°/N).
  - For each angle, convert to radians for math: θ = angle_i * (Math.PI/180).
  - Compute (x, y):
  - x = centerX + radius * cos(θ)
  - y = centerY + radius * sin(θ)
  - We then position the chip's center at (x,y). If using absolute positioning, we set:

```
.menu-ring__slot {
  position: absolute;
  left: xpx;
  top: ypx;
  transform: translate(-50%, -50%); /* to center the button at that
point */
}
```

    - Alternatively, we could use CSS transform on parent: an approach is to place each chip in HTML as if at center, and then do `transform: rotate(angle) translate(radius) rotate(-angle)` to spin them around (common trick for menu rings). But it might be simpler to just compute pixel positions as above due to dynamic content.
- Avoiding overlap: If N is large, chips could overlap. We capped pinned at 5, but if collapsed many to circle, N could grow. Realistically, if many panels minimized, the user might switch to desk mode. We might enforce that at most ~8 chips go to radial; beyond that, push additional ones to desk bar or another ring.
- If N <= 6, one ring is fine. If N ~7-10, chips will be a bit closer. If >10 (unlikely given pinned limit and typical usage), we should consider using the desk bar for overflow or using two concentric circles. For now, we assume N won't be huge (specifically pinned <=5).
- We also need to ensure chips don't collide with the **shell toggle button** when shell is hidden. That button might be at a corner, not near the ring, so fine. Or with other HUD like the restore button (the old `membraneRestoreButton` at bottom left [53], which likely will be removed or repurposed by us).

- We will include a margin from screen edges if chip would go off-screen (if center + radius close to edges). Possibly reduce radius slightly or ensure angle offset such that none is at extreme right if something there. But given symmetrical around center, should be okay.

- **Chip Hover Interaction:**

- When the user hovers over a chip (desktop), we will **expand** it. Expansion can be:
    - Increase scale (e.g. transform: scale(1.2) on the button).
    - Fade in or unhide the label text (`.menu-ring__label`) for that chip. We can have the label either appear as a tooltip above/below the chip or as part of chip widening.
    - One idea: the chip might by default be a circle icon only. On hover, it could sprout a little text label to the outside of the ring (e.g., a small tag with the panel name). Alternatively, the chip itself could morph into a pill shape containing the icon and name.
    - Simpler: we position the label element absolutely relative to the chip (like above or outside) and just toggle visibility on hover. For example, for each chip we could position the label on the radial line outward or inward. But since chips are around a circle, placing label consistently (like always outward of circle) might be best to avoid covering donut:
    - For angles in top half (approx -90° to 90°), place label above the chip or outside (which is upward).
    - For bottom half, place label below.
    - Or easier: always radial outward from center. Compute label position by extending a bit further out (like at radius * 1.1).

    - However, given few chips, we can probably just position them nicely by manual CSS or allow overlap if names are short.
    - We will implement a basic approach: each chip contains a label span which is `display: none` initially; on `.menu-ring__slot:hover .menu-ring__label { display: inline; }` and maybe increase parent z-index so it overlaps others if needed.
    - We also add a smooth transition for appearance (fade/slide). For instance, label can have `opacity:0; transform: translateY(5px);` normally, and on hover of parent, set `opacity:1; transform: translateY(0);`.

- Highlight: The chip's appearance could also highlight on hover (change background color or glow). We'll likely use the accent color (e.g. var(--accent) which is a gold) as a glow or border on hover to indicate it's active.

- **Chip Click Interaction:**

- Clicking a chip will **toggle** the corresponding panel:
    - If the panel is currently closed, we open it (floating window). We also possibly hide or mark the chip as active. Options:
    - Hide the chip while its panel is open to avoid duplicate representation (especially if panel might cover it).
    - Or leave it but indicate active (e.g. a different color). The advantage of leaving it is user can click it again to minimize/close. This double-click behavior might be handy. We will implement it such that clicking an inactive chip opens the panel; clicking the chip again (while the panel is open and not minimized) will collapse it back. This means we need to know if that panel is open. We do: in state or by checking DOM (panel has no hidden attribute).
    - If panel is open and user clicks chip:
    - If the panel is not already minimized, we interpret that as a request to minimize/close it. Possibly collapse it (rather than full close) is nicer (so it goes back to chip form cleanly). But since they clicked the chip (which is essentially the collapsed form), them clicking while it's open might not happen because if panel open we might hide chip or mark it differently.
    - Perhaps a better approach: when panel opens, we **temporarily hide its chip** (to reduce clutter). Or disable it.
    - Then to minimize, user would use the panel's collapse button.
    - This avoids confusion of chip click while panel open.
    - We lean to: hide chip when panel open. Then closing/collapsing panel makes chip reappear. This also visually links the idea that the chip "became" the panel and is not available separately until you send it back.
    - Implementation: when `openPanel(panelId)` is called from a chip, we can add `chipElement.style.display='none'` or add a class `.is-active` that maybe sets visibility hidden. And when panel is closed or collapsed, remove that.
    - If multiple chips, this ensures the one in use is out of the way or clearly inactive.
    - If panel is minimized (circle mode state), clicking chip will restore it (which is consistent: that chip essentially is the panel's representation).

- We will also ensure focus styles for keyboard: user can Tab to chips (they are buttons). Pressing Enter/Space on a focused chip triggers the same open/close logic.

- **Active State Indication:**

- For chips corresponding to open panels, we should indicate that state (especially if we don't hide them). Perhaps by color change or a filled dot.
- Maybe use the status dot concept on chips too: e.g., a small dot on the chip if the panel's process is running (like EEG streaming).
- But the chip icon could itself be considered a status indicator (like Brain Waves chip could pulse).

- We'll implement a simple highlight ring on active chips (like a glowing border) to denote "this panel is currently open". If we hide the chip entirely when open, then we don't need that, but hiding might be jarring. We could try both approaches during testing.

- **Dynamic Updates:**

- If the user pins or unpins an item while shell is open, we need to update the radial menu in real-time:
    - Pinning an item adds it if shell is hidden or circle mode on. We can just always keep radial updated even if shell visible (chips can sit there even if shell shown in overlay, not a big problem, but we might hide radial UI when shell overlay is open to reduce distraction).
    - Conversely, unpinning removes its chip (unless it's currently open and minimized specifically).
- If user collapses a panel to circle, we add chip.
- If user opens a panel that had a chip, remove/hide chip.

- We might write a `renderRadialMenu()` function that looks at `state.ui.pinnedPanels` and `state.ui.openPanels` and builds chips accordingly. But doing incremental updates on each action might be simpler and less performance overhead given few items.

- **Visual Style:**

- Chips will be styled as circles. Use a CSS class like `.menu-ring__slot` on the button to give it:

```
width: 48px; height: 48px; border-radius: 50%;
background: rgba(255,255,255,0.1); /* translucent */
backdrop-filter: blur(3px);
border: 2px solid var(--accent-soft); /* a soft colored border maybe */
color: #fff;
display: flex; align-items: center; justify-content: center;
transition: transform 0.2s, background 0.2s;
```

On hover or focus, we change:

```
.menu-ring__slot:hover { background: rgba(255,255,255,0.2); transform:
scale(1.1); border-color: var(--accent); }
```

- The icon inside can be a font icon or emoji, we'll style `.menu-ring__icon { font-size: 1.2em; }`.

- The label `.menu-ring__label` we might position absolute relative to the chip or simply hide/show inline. Possibly better to position absolute so it doesn't change layout:

    - One approach: put the label in a sibling container to the ring (complicated).
    - Simpler: when label is shown, we allow it to be position relative to chip: e.g.

      ```
      .menu-ring__slot { position: absolute; }
      .menu-ring__label { position: absolute; white-space: nowrap; font-
      size: 14px; background: #000A; padding: 2px 6px; border-radius: 4px;
                          opacity: 0; transition: opacity 0.2s; pointer-
      events: none; }
      ```

      and depending on chip's quadrant relative to center, we set label's placement: We can dynamically set a data attribute like `data-angle` on each chip. Then CSS can use that to decide:
    - If angle near top: place label above (bottom: 120% perhaps).
    - If bottom: place above? Actually if bottom quadrant, place label below (top: 120%).
    - If left vs right: align text accordingly (maybe left of chip vs right of chip). This might be too much for CSS alone; easier might be computing explicit label coordinates along with chip coordinates. Simpler alternative: just make the chip bigger and show text next to icon within an elongated button. However, that could overlap other chips.
    - Perhaps the straightforward method: reveal label as a tooltip is fine. We'll do some basic: if angle <= 90 or >= 270 (top half) show label above, else below; if angle between 45 and 135 (right-ish) show right, etc. We'll implement minimal JS to position a tooltip div near the chip on hover (like a classic tooltip approach). Given time, we might skip perfect placement and just show a centered label above the chip always. The user can still figure it out, especially since at hover they know which one they aimed at.
    - For now: on hover, we can simply set `.menu-ring__label { display: block; opacity:1; position: static; margin-top: 4px; }` and it will show below the chip in flow, but since chip is absolutely positioned, position: static would actually place label in button which might not enlarge properly. Possibly make button container bigger to accommodate label text? That would disrupt circular arrangement though. So scratch that.
    - We'll likely do an absolutely positioned tooltip element outside the chip. Perhaps simpler: use the `title` attribute on the button so the browser shows a tooltip on hover. This is the simplest fallback (though not as stylish). Actually using `title` might conflict with custom styling but it's a quick accessibility win. We can do that for fallback anyway, and later add fancy labels.

- Considering scope, using the native tooltip via `title` might be enough for showing names on hover. The prompt did mention "hover/focus expands, click opens" implying a custom expansion UI, so we will include the label in DOM but if time is short it can be non-animated or even hidden behind needing user to hold hover momentarily. But let's implement the label as above and tune placement minimally.

- **Performance:**

- The radial layout and small number of elements means this will be lightweight. We just need to update positions on window resize (in case size changes, we recalc center). We can use a ResizeObserver or window.onresize to re-layout the chips.
- If the donut's camera moves or rotates (does it? If user has orbit controls to spin donut, the chips are screen-space static, not following 3D rotation of donut since they are UI). That's fine—they are anchored to screen, not geometry. If we wanted them to literally follow donut rotation (like anchored in 3D space around it), that would be more complex (project 3D positions to 2D each frame). That's probably not intended; likely these chips are pure overlay UI fixed relative to screen (which is simpler and what we'll do).

In summary, Circle Mode creates a visually striking **ring of quick-launch icons** around the donut, matching the aesthetic of a futuristic HUD. It leverages the pinned panel concept to decide what appears, ensuring the ring isn't cluttered. The algorithm places items evenly, with intuitive hover expansions. This mode is especially useful during "immersive" use when the directory is hidden – users can still trigger panels without breaking immersion. It aligns with the concept of "spatial menus anchored to geometry" mentioned in the dev notes [54].

Desk Mode Specification & Algorithm (Bottom Dock)

Desk Mode is an alternative minimization mode where panels are represented as icons in a dock at the bottom of the screen (analogous to a desktop taskbar). It's suited for scenarios with many panels or where a horizontal layout is preferred. Below we outline the desk mode UX and the logic for managing the dock.

- **When Desk Mode is Used:** Desk mode can be the primary minimize strategy if:
- The user has more panels minimized than can comfortably fit around the donut, or
- The user explicitly prefers it (maybe via a user setting or simply by collapsing panels when shell is docked).
- For our implementation, we might decide that if a panel is not pinned (no dedicated radial spot), collapsing it goes to the desk bar. Also, if more than ~5 items are minimized in total, all go to desk (to avoid a very crowded circle).

- We could also allow a manual toggle: e.g., a small button to "switch to desk mode" which would move all circle chips down to the dock. The spec suggests supporting both, possibly simultaneously (pinned in circle, non-pinned in desk). This hybrid approach might be best:

    - **Pinned favorites** continue to appear around donut (since max 5, nice and visible).
    - **All open panels** (including pinned ones if open?) appear in desk bar as well, or maybe only non-pinned open ones.
    - Alternatively, treat desk bar as **open windows bar**: whenever a panel is open or minimized, it has an icon on the desk bar. Pinned vs not doesn't matter for the bar; pinned only influences radial presence.
    - This way, desk bar works like a typical OS taskbar: each open panel is represented. If closed, remove from bar (unless pinned scenario).
    - And radial ring works like quick launch for pinned items (open or not).
    - This interpretation fits the "dense sets" idea: if user opens many panels, the bottom bar will fill with their icons (dense but scrollable), while the radial stays limited to favorites. We'll adopt this model.

- **Desk Bar Structure:** We will create a container `<div id="deskBar" class="desk-bar"></div>` likely positioned at bottom center of screen. CSS:

```
.desk-bar {
  position: fixed;
  bottom: 16px;
  left: 50%;
  transform: translateX(-50%);
  display: flex;
  flex-flow: row nowrap;
  align-items: center;
  background: rgba(255,255,255,0.1);
  backdrop-filter: blur(5px);
  padding: 4px 8px;
  border-radius: 12px;
  max-width: 80%;
  overflow-x: auto;
  scrollbar-width: none; /* hide scrollbar */
}
.desk-bar::-webkit-scrollbar { display: none; }
```

  This creates a translucent strip (slightly blurred background) with icons inside. We center it, but if it's very wide it can extend towards edges (80% max width and scroll if needed).

- We ensure it has pointer-events auto so it's interactive (since floatingDockLayer was pointer-events none globally, we might want to either have deskBar outside that or override pointer-events on it).
- It's `aria-live="polite"` (maybe, to announce changes like "3 windows open").
- **Desk Icons:** Each panel that is open (in any state except fully closed) gets an icon button in the desk bar. Similar to radial chips but smaller, maybe square-ish icons:

```
<button class="desk-bar__icon" data-panel-id="unitCirclePanel" title="Unit
Circle">
  <span class="desk-bar__icon-img">◯</span>
</button>
```

- Styling:

```
.desk-bar__icon {
  width: 40px; height: 40px; margin: 0 4px;
  border: none; background: rgba(255,255,255,0.15);
  border-radius: 8px;
```

```
  display: flex; align-items: center; justify-content: center;
  color: #fff;
  position: relative;
}
.desk-bar__icon-img { font-size: 1.2em; }
.desk-bar__icon:focus { outline: 2px solid var(--accent); }
.desk-bar__icon--active { background: rgba(255,255,255,0.3); } /* for
active (focused) panel */
```

- We can use the same icon imagery as radial (maybe the exact same emoji or letter). Possibly the desk icons could be slightly smaller or omit text entirely.
- We may overlay a small indicator dot on the icon if the panel is running some process or to indicate it's open vs minimized:
    - If we want to show which icons correspond to currently *open & visible* vs *minimized*, we could show e.g. a filled dot for open windows and a hollow dot for minimized. Or a different opacity.
    - Or simpler: if a panel is minimized (circle or desk), the icon is normal; if the panel's window is currently open (not minimized), highlight the icon (e.g. brighter background or a small "window" indicator).
    - This helps the user know which tasks are currently on-screen.
    - We could use `.desk-bar__icon--open` class for open ones.
    - Alternatively, since if panel is open it might not be in desk bar if we remove it when open? But we intend to keep it to manage windows. I think keep it (like Windows taskbar keeps icon even if window open).
    - So yes, we'll highlight open ones. When user collapses it, the highlight goes away (since now it's just in bar as minimized).
- The `title` attribute on the button gives the name on hover (so user can identify if icon is just an emoji/letter).

- If many icons overflow width, user can scroll the bar horizontally (we hide scrollbar for aesthetics but allow scroll via trackpad or mousewheel on bar). We could also implement arrow buttons to scroll, but maybe unnecessary if <= 10 or so.

- **Behavior on Click:**

- Clicking an icon toggles that panel's state:
    - If panel is minimized (either to bar or radial, but if it has an icon it's effectively in bar now), clicking icon will restore/open the panel (floating).
    - If panel is already open and visible, clicking its icon could either *minimize* it (hide it back to bar) or bring it to front if it's behind others.
    - Many OS docks minimize on click if already focused; others just indicate focus. We have a collapse button in panel UI for minimizing explicitly. Perhaps we do:
    - Left-click on icon:
        - If panel not currently focused front, bring it to front (focus it).
        - If it is already the frontmost focused panel, minimize it (toggle).
    - This gives a nice toggle behavior but might confuse if mis-click. But advanced users expect click to minimize if already active.

    - Alternatively, always toggle minimize (like macOS: clicking dock icon always brings forward, and to minimize you must explicitly click minimize in window, not via icon).
        - Actually on macOS, clicking an open app icon that has multiple windows just brings them front, it doesn't minimize. On Windows, clicking taskbar toggles minimize for single-window apps. It's inconsistent across OS.
    - We need to choose. Given we have a separate collapse control, we might mimic Mac: clicking icon always shows the window (if hidden or behind others). To minimize, user should use collapse control or right-click context (which we might not have).
    - So we will implement: clicking desk icon **always ensures the panel is visible and focused** (so unminimize if needed, and bring front). If already front, clicking does nothing (we could implement double-click to minimize, but not needed).
    - We will include a right-click context menu idea in "cool ideas" perhaps, but not implement now.

- Example:

    - Suppose *Unit Circle* panel is minimized (icon in bar). User clicks icon: panel appears (floating) at last known position (or center if lost) and is focused (z-index top). The icon now is highlighted indicating open.
    - If user clicks another icon of *Field Deck* while Unit Circle open: Field Deck panel either appears or is brought front. Unit Circle panel remains open but behind (unless user collapsed it).
    - Now if user wants to quickly hide Unit Circle again, they can either click its collapse button or possibly click its dock icon again (depending on our rule). With our chosen rule, clicking again will just bring it front (already is, so no change). So they should use collapse button.
    - This is fine; it mirrors macOS behavior somewhat, which is generally okay for simplicity.

- **Adding/Removing Icons:**

- When a panel opens (from any trigger), we add an icon to the bar if not already present. Likely if panel is pinned and opened via radial, it won't have been in bar if previously closed, so we add it now.
- We might actually pre-add pinned ones? Probably not; desk is for open tasks primarily, not for all pinned if closed (that's radial's job).
- When a panel is closed completely, we remove its icon from the bar.
- When a panel is collapsed/minimized, we keep its icon (it's still "open" in the sense of loaded, just hidden).
- If a user unpins a panel while it's still open, that doesn't affect bar (bar doesn't care about pinned).
- If a user pins a panel that's open, that just adds radial icon, bar icon was already there from open.

- We ensure the bar always only shows unique panel icons; no duplicates.

- **Indicator for Panel Status:**

- Similar to radial chips, we might incorporate small status lights on the dock icons if relevant. For example, a tiny colored dot at bottom-right of icon indicating if something is active. E.g., for Brain Waves, dot green if streaming.
- This can be done by adding a child element `.desk-bar__status-dot` inside the button, absolutely positioned bottom-right. We can update its class when panel status changes. This is a

nice-to-have for feedback (not explicitly asked but matches “status dot” concept). We already have status dot in panel header and in list, adding it on icon too would unify.

- Prioritize: If time, yes; if not, skip because not in spec explicitly.

- **Scrolling Behavior:**

- If the bar has more icons than can fit, the user can scroll it. We’ll enable horizontal scrolling by mouse wheel (listen for wheel events and translate horizontally) for convenience. Many modern UIs do that for tab strips etc.
- Or just let them drag-scroll (we can set `deskBar.style.cursor = 'grab'` and implement pointermove to scroll).

- For now, basic overflow-x auto with trackpad is sufficient.

- **Desk vs Circle Coordination:**

- As mentioned, pinned favorites typically go to radial; everything open goes to desk.
- If a panel is pinned and open, it will appear both as radial (pinned chip) *and* on desk (open window).
- That’s fine – radial chip can be hidden if open (if we do that), or remain but user sees two places.
- Might be slight duplication, but they serve different purposes: radial chip to open it initially (since it’s a favorite), and dock icon to manage it as a window.
- We can live with that because advanced users will understand one is a launcher, one is a task indicator. Or we could hide radial when open as earlier plan, which avoids duplication.
- We will implement the hiding of radial chip for open pinned panel to reduce redundancy (so a pinned panel’s chip disappears when it’s open, reappears when closed). This way at any time, a given panel is either represented by a big window or by a chip, but not both (except in desk which always shows open windows).
- That means for pinned panels, when closed they show in radial, when open they move to desk (with icon highlight).
- For non-pinned panels, when closed they are nowhere, when open they show in desk (and maybe if minimized remain in desk).

- This sounds consistent.

- **Keyboard and Accessibility:**

- Users can use Tab/Arrow to focus icons on the desk bar. Pressing Enter triggers the click logic (open/focus window).

- We might assign shortcut keys to directly switch panels if needed (like Alt+Tab – but that might conflict with OS). Not required now, but maybe mention in extras if relevant.

- **Desk Mode Activation (if any explicit control):**

- The spec hints at possibly toggling between circle mode vs desk mode. If we interpret differently, maybe they wanted a user option like “Bundled desk mode” vs “Radial mode” for all minimized panels.

- It's possible the user might not want radial at all and prefer everything on a bottom bar (some users might find radial fancy but not practical).
- We could have a toggle in UI (maybe in the shell header or in a settings panel) to choose "Use radial menu for favorites [on/off]" or "Circle mode vs Desk mode".
- However, the question phrased them as separate specs, which we are fulfilling by implementing both concurrently as above. It doesn't explicitly say user toggles between them globally.
- We will default to using both: pinned to circle, open windows to desk. This covers both scenarios.
- But to strictly follow structure, maybe we can describe "Circle Mode spec" and "Desk Mode spec" as if either could be the chosen style.
- If the product owners want a single mode at a time, we could implement a global flag `state.ui.interfaceMode = 'circle'|'desk'` that determines how all minimizations behave.
- But since we see synergy in using both, we'll present it as a combined approach in practice. We can still note that an option can switch fully to desk mode (meaning pinned items also show in desk bar even when closed, like a traditional taskbar with pinned apps).
- If that were needed: we'd simply create desk icons for pinned items even when closed (like pinned apps in Windows). That's feasible but not requested.
- We will mention as an "open question" possibly whether to allow switching off radial.

With Desk Mode in place, the UI gains a familiar anchor point for multitasking. Imagine the user has collapsed several panels: instead of hunting for them, they see a row of icons at the bottom – e.g., a crown icon for EEG, a donut icon for Everything, a gear for settings, etc. They can click to recall any panel. This mode complements Circle Mode: circle for quick launching favorites, desk for managing many open panels. The algorithm and design are guided by conventional taskbar patterns, ensuring the user isn't overwhelmed even with "dense sets" of panels.

Migration Steps (Implementation Plan Checklist)

To transition from the current codebase to this redesigned system, we will follow an ordered series of steps. This ensures we preserve functionality while incrementally integrating the new design. Below is the checklist of actions, roughly in sequence:

1. **Backup & Analyze Current Sidebar/Panel Code:**
2. Before making changes, document the existing sidebar mechanics and panel structure. (The dev journal notes from entries #017–#024 have snapshots of old behavior [55] [56], which we've used to plan removal of ghost/pinning overlays.)
3. Confirm which elements and functions from old system remain relevant. E.g., `#sidebarShell`, `#sidebarRight` (ghost) – ghost is no longer needed, and `pinMainMembranePanel` is stubbed [57], etc.
4. We will remove or disable the now-obsolete constructs:
    - The right ghost sidebar (`#sidebarRight` in HTML) and any references to it in CSS/JS. (It's already `aria-hidden="true"` and collapsed via CSS [58], so simply ensure no JS tries to use it.)
    - Old pinning logic that assumed docking left/right. (Functions like `attachMembraneDirectoryToShell()` remain for directory injection, but things like `dockMembranePanelLeft/Right` if any, or the slider UI for pin left/float/right in `initDockablePanel` will be refactored.)

- The `PIN_SIDEBAR_MODES` remains (we keep Hidden/Overlay/Docked) but any references to multiple sidebars or ghost elements can be excised.

5. Essentially, create a clean slate for the sidebar shell, treating it as a single container.

6. **Implement Shell (Sidebar) Redesign:**

7. **HTML Adjustments:** Edit `index.html`:
   - In `#sidebarShell` aside, ensure the structure has:

     ```
     <div class="sidebar-shell__header">
       <div class="sidebar-shell__title">Membrane</div>
       <button type="button" class="sidebar-shell__toggle"
     id="sidebarShellToggle" aria-pressed="false">Hidden</button>
     </div>
     <div class="sidebar-shell__body" id="sidebarShellBody"></div>
     ```

     If not already present (based on snippet 59 it is). The toggle's text we might initialize to "Hidden" or actual current mode.
   - If there's leftover code for a topbar or overlay, remove it (the journal mentions removing scrim/overlays earlier).
   - Add placeholder elements in the body for filter and sort controls:

     ```
     <div class="membrane-controls">
       <input type="search" id="membraneSearchInput"
     placeholder="Search..." />
       <div class="membrane-filters"> ... (filter buttons) ... </div>
       <select id="membraneSortSelect">...</select>
     </div>
     <div class="membrane-directory__list" id="membraneDirectoryList"></
     div>
     ```

     Actually, the existing structure had a search and a list wrapper 22 60 ; we will repurpose those:
   - The search input is already there; we will just add filter buttons below it in the DOM.
   - The sort dropdown could be added next to the search input or above the list. For UI, perhaps align sort to right of search bar. We can wrap search+sort in one flex container.
   - Add a section title for pinned if desired: we can generate it in JS instead of static HTML.

8. **CSS Updates:** Modify `light.css`:
   - Ensure `.sidebar-shell.is-overlay`, `.is-docked`, `.is-hidden` rules exist (they do 61). These control display. We might adjust `.is-overlay` to maybe have a max-height or width if needed. Also verify z-index layering (sidebarShell should be above canvas, likely z-index 30 as .sidebar suggests 62).
   - Style new elements:

- `.membrane-controls` (search/filter/sort container) – maybe some margin-bottom.
- `.filter-btn` styles: make them togglable (e.g., background color change when active).
- `.membrane-directory__section-title` if used for "Pinned".
- Hide default list bullets if any (we use divs, so fine).
- We likely need to remove any old background from sidebar: The dev journal mention making panel backgrounds transparent [63]. Check that `--panel-bg: transparent` is set (yes at root [21]). If any `.sidebar` or `.sidebar-shell` backgrounds remain, adjust to none.
- The shell header style: It likely has minimal style now (we can add maybe a bottom border line or subtle shadow to delineate). Ensure `sidebar-shell__title` and `__toggle` are styled consistently (the snippet shows them white text on violet background perhaps [64]).
- Inset scrollbars: The CSS snippet shows attempts to style `.sidebar` scrollbars [65]. We might adapt those for `sidebar-shell__body` or use the same if it inherits `.sidebar` class (currently `aside.sidebar-shell` has no `.sidebar` class, so those rules might not apply – we should duplicate scrollbar styling for `.sidebar-shell__body`).
- Add a rule to hide the shell toggle button when shell is hidden and empty: e.g.,

```
#sidebarShell.is-hidden[data-empty="true"] .sidebar-shell__toggle {
display: none; }
```

  We will set `sidebarShell.dataset.empty = "true"` via JS when appropriate.
- Remove any leftover `.sidebar--ghost` rules since ghost aside gone.
- Test visually: in overlay mode, shell appears above donut with proper size; in docked, it's left aligned.

9. **Build Directory List Rendering (JS):**

10. In `app.js` or a new module (e.g., `src/ui/shell.js`), implement the logic to populate the directory list. We can replace or extend the existing `renderMembraneDirectory()` [38]:

    - Integrate filter and sort:
    - Precompute category mapping for panels (maybe a constant object or derive from `data-membrane-category` attributes we add to HTML for each panel `<section>`).
    - Filter panels by `state.ui.filters` and search term (use `.includes` on name).
    - Sort the filtered list according to `state.ui.sortMode` (alphabetical by name default, or by lastOpened times).
    - Split into pinned vs others:
      - `pinnedPanels = state.ui.pinnedPanels` (array). Iterate pinned in that order:
      - For each, if it exists in all panels list and passes search/filter (we may decide pinned should always show regardless of filter? Possibly not; filter might hide pinned if category off. Probably filter applies globally including pinned).
      - Generate its item (with Unpin and Open actions).
      - Then for others, exclude those pinned and generate list items similarly.
    - The existing code already filters out `data-membrane-fixed="true"` panels (like Entry Door and Directory itself) [26]; we keep that to exclude control panels from listing.

- Update action buttons: use Pin/Unpin and Open as defined, instead of Pin L/R, etc. This means adjusting the `actionMeta` creation in the loop [32]. Remove references to dock-left/right and float.
- Ensure to mark disabled if needed (for pin when at max).
- Append section headers: if we have any pinned items and we want a label, we can prepend a small `div.memebrane-directory__section-title` with "Pinned (n)" to `membraneDirectoryList`.
- After building items, call `sidebarShellBody.appendChild(listFragment)` (the code currently appends to `membraneDirectoryList`).
- *Note:* The old code attached the directory panel into the shell body via `attachMembraneDirectoryToShell()` at the end of initialization [66]. After migration, we might not need a separate `#membraneDirectoryPanel` at all – the shell *is* the directory container. Indeed, currently `membraneDirectoryPanel` is a section in floating-layer that gets moved inside shell. We can simplify by directly using shell's body for list instead of moving that section. So:
- Option A: Continue using `#membraneDirectoryPanel` as the container for list content, but inside shell.
- Option B: Eliminate `#membraneDirectoryPanel` section entirely, and just build the list in shell from scratch (preferred to reduce complexity).
- We likely choose B: Remove the `<section id="membraneDirectoryPanel">` from HTML (or leave it hidden as backup), and let the aside's body hold the directory UI. The references in code to `document.getElementById('membraneDirectoryPanel')` can be removed (like in attachMembraneDirectoryToShell).
- This aligns with dev note of simplifying to "transparent shell with membrane list/actions" [56].
- Remove or refactor `attachMembraneDirectoryToShell()`: it appended the existing directory panel into shell and set overlay mode [66]. With our approach, we can just ensure shell is visible by default in overlay after initialization (maybe call `setSidebarMode('overlay')` once panels are built, similar to what attach did [67]).
- Setup event listeners:
- Search input `oninput` to update `state.ui.searchQuery` and call render.
- Filter buttons `onclick` toggling classes and updating `state.ui.filters` then render.
- Sort select `onchange` update `state.ui.sortMode` and render.
- Directory list actions (pin/open) via event delegation (we keep `membraneDirectoryList.addEventListener('click', ...)` as the code had [34], adjusting actions).
- Possibly add double-click on an item row to open too (nice to have).
- Mark shell as empty or not: after rendering, if no list items were rendered (i.e., no membranes available, which could happen if filters exclude all), set `sidebarShell.dataset.empty="true"` to possibly hide toggle. If renderedCount > 0, remove that attribute.
- Testing: After implementing, test by calling `renderMembraneDirectory()` manually (or after state setup) to see if the list populates correctly with current panels.

11. **Panel Header & Controls Integration:**

12. Modify each panel's HTML structure in `index.html`:
    - Go through each `<section class="panel membrane-panel" id="...">`:

- If it doesn't have a `<header>` element containing the title, create one.
  - Example: for `everythingPanel`, currently it has `<h2 class="panel__title">The Everything Chalice</h2>` as direct child [68]. Wrap that in `<header class="panel__header"><div><p class="panel__meta">...</p><h2>...</h2></div> ...</header>`. Actually many panels have a similar structure with meta and title in a header container already (like we saw `lolPanel` has a header with title and a toggle [69]).
  - So for consistency, ensure a `<header class="panel__header">` exists in each. If a panel uses a different class, unify it.
  - If there are subtitle or meta lines, keep them within the header (perhaps in a `<div>` with title).
- Add the controls container in the header. E.g.

```
<header class="panel__header">
  ... existing title/meta ...
  <div class="panel__controls">
    <button type="button" class="panel-btn panel-btn--pin"
title="Pin panel"></button>
    <button type="button" class="panel-btn panel-btn--focus"
title="Focus panel"></button>
    <button type="button" class="panel-btn panel-btn--collapse"
title="Collapse panel"></button>
    <button type="button" class="panel-btn panel-btn--close"
title="Close panel"></button>
  </div>
</header>
```

  We could use innerHTML or DOM creation in JS to append this to each panel if doing manually is cumbersome. But since the user provided the code, manual edit is possible. We have to be careful not to break panel-specific controls (like brainwave's existing collapse). For brainwavePanel, we'll remove its custom collapse button in favor of ours. For others, just adding our controls.
- Optionally add `span class="panel__status-dot"` in header next to title if we want. Could do `<h2>Title <span class="panel__status-dot"></span></h2>`.
- Ensure each panel has `data-membrane-name` attribute (most do) for display name and maybe add `data-membrane-category` attribute in HTML according to our mapping (so filter can read it). If not adding in HTML, we'll hardcode mapping in JS. Adding in HTML is straightforward:
  - E.g. `<section ... data-membrane-category="Visual">`. We'll do that for all panels in index.html.
- CSS: define `.panel__header { display:flex; justify-content: space-between; align-items:center; padding: 8px; }` possibly. Style `.panel__controls button` as per earlier description. Possibly use small icons (we can use Unicode emoji for now: pin , focus 🔍 (or an eye), collapse minimizing icon like an underscore or 🗕, close ✖). If using Unicode, set as button text or ::before content. Or include an icon font (lightweight one).

- Given no external library call, maybe Wingdings or just letters might be unprofessional. We could embed SVGs for these common icons:
  - Pin: maybe we have none readily, fallback to simple text "Pin".
  - For clarity, maybe use letters [P][F][–][X] as placeholders (not ideal).
  - Actually, we can use the same ghost button styling (which currently likely expects text as it had "Pin L", etc.).
  - Possibly load an icon set like Feather icons via JS later, but to keep it simple, use text "Pin"/"Unpin", "Focus", "Minimize", "Close".
  - We'll style them as small ghost buttons (the CSS for .ghost-btn can be reused for .panel-btn, or just use ghost-btn class on them). The existing code uses ghost-btn for directory actions [70] .
  - Maybe use ghost-btn style for these controls as well so they inherit a subtle style, or define new style if needed (maybe icons look better).
- We'll refine after basic functionality works.

13. **JS Behavior for Panel Controls:**
    - In the initialization script (after panels are created in DOM), attach event listeners for each control button. Could do via event delegation on panel container or individually:
    - E.g.

```
document.querySelectorAll('.panel-btn--pin').forEach(btn => {
   btn.addEventListener('click', () => {
      const panel = btn.closest('.membrane-panel');
      togglePin(panel.id);
   });
});
```

    Similarly for focus, collapse, close:
    - `focus`: bring that panel to front (`bringToFront(panel)`).
    - `collapse`: call `collapsePanel(panel.id)` which will hide panel and either call `moveToCircle(panel)` or `moveToDesk(panel)` based on conditions.
    - `close`: call `closePanel(panel.id)`.
    - Implement these helper functions:
    - `togglePin(panelId)`: if panelId in state.ui.pinnedPanels -> remove it (unpin), else if length < max -> add it. Then call `renderMembraneDirectory()` to update list UI. Also, if pinned and shell is hidden, perhaps create radial chip (we can in togglePin also call an update to radial menu).
    - `bringToFront(panel)`: sets a high z-index or moves node to end of floatingLayer. (We have `floatingLayer` div; maybe easier: `floatingLayer.appendChild(panelEl)` will move it to end). Also mark this panel as focused (maybe add class .is-focused, remove that from any other).
    - `collapsePanel(panelId)`:
      - Identify panel element. Remove its `.is-hidden` if any (should already be visible if collapsing? Actually we collapse visible ones).
      - Determine target mode: If `state.ui.circleModeEnabled` (if we had such a flag or if panel is pinned?), or if pinned, go to circle. Otherwise, go to desktop. We'll do: if panelId in pinnedPanels -> to circle, else -> to desktop.

        - For circle:
        - Add class `.membrane-panel--circular` to panel element (for state reference).
        - Actually hide the panel's DOM: we could do `panel.hidden = true` so it's not visible, but keep in DOM (so we can bring it back easily and preserve state).
        - Create a radial chip for it (if not already one). Actually pinnedPanels likely already have a chip. But if user collapsed a non-pinned panel and we decide to allow circle for it (maybe not, maybe all non-pinned go desk).
        - We might restrict circle to pinned only, to avoid unpredictable many chips. So likely: if not pinned, we won't collapse to circle, we do desk. So collapsePanel logic simplifies: pinned item -> hide panel (so effectively it's now only accessible via its pinned chip, which we ensure is present; set state panelDockState to circle).
        - For desktop:
        - Add class `.membrane-panel--desktop` to panel element.
        - Hide the panel (panel.hidden = true).
        - Ensure there's an icon in desk bar (if not, create one).
        - If panel was focused, maybe focus moves somewhere else (like to document or to next panel).
        - In both cases, update `state.ui.openPanels` entry for that panel's mode (or remove from openPanels? Actually no, still open just minimized).
        - Also, if panel was currently focused top, maybe set focus to none or next panel behind (but not critical).
    - `openPanel(panelId)`: (used when clicking a radial chip or directory open action)
        - Find panel element. If it's already in `openPanels`:
            - If it's minimized (hidden):
            - Show it (panel.hidden=false, remove .is-hidden class).
            - If it had .membrane-panel--circular or --desktop class, remove those (because now it's not minimized).
            - If it was in radial menu and pinned, perhaps hide its chip (set style or a state to hide).
            - Bring to front (so it's visible above others).
            - Focus it (maybe focus first input or add .is-focused style).
            - Update state (mark as not minimized perhaps).
        - If it's not open yet:
            - Append it to floatingLayer if not already.
            - Remove hidden attribute.
            - Position it (center or maybe last known if we have state).
            - Add to state.ui.openPanels.
            - Create desk icon for it (because now it's open).
            - If it's pinned, also possibly hide its radial chip as discussed.
        - Possibly call some panel-specific init if needed (maybe not, since app already did initial init).
    - `closePanel(panelId)`:
        - Hide panel (panel.hidden = true, add .is-hidden).
        - Remove from state.ui.openPanels.
        - Remove any minimized state classes if present.
        - Remove desk bar icon.
        - If it's pinned, ensure a radial chip exists (unhide chip if it was hidden during open).

        - If it's not pinned, it will now not be accessible anywhere (except via directory list).
        - If closing last panel, maybe do some cleanup (like if none open, you might re-show something or free resources).
        - If the panel has any active processes (like streaming), optionally stop them here by calling its teardown if available.
    - Note: some of these functions overlap with existing ones in code (like maybe they have something to open panels). We should either integrate or replace them carefully. For example, current code might open panels on certain events (like Entry Door route selection opens a sequence of panels, etc.). We need to adapt those to use our new openPanel logic.
        - We should search for places in `app.js` that call `document.getElementById(panelId).removeAttribute('hidden')` or similar to show panels, and replace that with openPanel call or incorporate our new needed steps (like adding desk icon).
        - Possibly, the entire app relied on toggling `hidden` attribute and some state arrays. Now we have more apparatus, so we might catch common triggers:
        - The "Enter the membranes" button likely opens a route of panels (the route from Entry Door panel gave an array and then presumably opened each).
        - We will override those specific flows to ensure they open via our system. For example, after route calculation, instead of just un-hiding the sections, we call openPanel for each in sequence or so.
14. Attach event listeners to radial chips and desk icons:
    - For radial chips, delegation on `menuRingEl` or add individually since number is small:
    - On click, call openPanel(panelId) for that chip (since radial chips represent closed pinned panels typically).
    - On hover, show label (we can handle via CSS for now).
    - For desk icons:
    - On click, implement focus behavior: if panel is not open (should always be open if it's there, unless we choose to keep icons for closed pinned which we are not), but anyway, if open and hidden (minimized), openPanel (unhide it). If open and visible, bringToFront.
    - Possibly on double-click (or right-click) we could close or other actions, but not in this scope. We keep simple.

15. Remove old event handlers:

    - E.g., the old code for `sidebarShellToggle` click uses `cycleSidebarMode()` [71] which is fine, we keep that.
    - Old code for pin buttons if any (like `pinMainMembranePanel`) – likely removed earlier.
    - Old code handling swipe to shelf: There might be some logic if panel swiped beyond threshold to shelf it. That is advanced; we might leave it disabled for now and add manual collapse only. If `DOCK_SWIPE_THRESHOLD` etc. exist [72], they may be related. Since we removed ghost/pinning overlays, probably that code is dormant. We won't implement swipe yet, could consider as extra idea.

16. **Implement Radial Menu UI:**

17. Add HTML container for radial menu:
    - Possibly in `index.html` at a convenient place, e.g., just inside `<div class="app">` or as sibling to floatingLayer. The dev notes mention "Membrane navigator merged into menu-

ring" [73], implying an element for menu ring existed. Search for `menuRing` earlier turned up none in HTML, meaning it might be created in JS or was meant to be added.
    - We will insert `<div id="menuRing" class="menu-ring" aria-hidden="true"><!-- chips will be inserted --></div>` near the end of the body (just above desk bar perhaps).
    - Or we can create it dynamically in JS if we prefer.
18. CSS:
    - `.menu-ring { position: fixed; top:0; left:0; right:0; bottom:0; pointer-events: none; }` to cover screen but ignore clicks except on chips as set pointer-events auto on them (we already have `.menu-ring__slot { pointer-events: auto; }` CSS).
    - style for `.menu-ring__slot` (done above).
    - Possibly a central circle indicator? Not needed, donut is visible.
19. JS:
    - Create a function `layoutRadialChips()` that:
    - Takes list of panel IDs to display (likely `state.ui.pinnedPanels` minus those currently open? Or plus any collapsed panels flagged to radial).
    - Calculate positions as per algorithm.
    - For each, either create or update an existing chip element.
        - We can maintain a map of chip elements by panelId for reuse, or reconstruct each time if N small (fine).
        - If constructing:

```
const btn = document.createElement('button');
btn.className = 'menu-ring__slot';
btn.dataset.panelId = id;
btn.title = panelName;
btn.innerHTML = `<span class="menu-ring__icon">${icon}</
span><span class="menu-ring__label">${panelName}</span>`;
menuRing.appendChild(btn);
```

        - Set its left/top via style or using CSS transform if we choose that route. Likely easier: style.left = (x)px; style.top = (y)px.
    - Use `pointer-events: none` on the menuRing container, but each button needs pointer-events auto or default (they are by default clickable). Ensure CSS covers that (we saw `.menu-ring__slot` not covered by pointer-events none due to it being child and override).
    - Attach event listeners to each button: click -> openPanel as discussed; focus/blur or mouseenter/leave can trigger label show if not done purely in CSS.
    - Mark menuRing `aria-hidden="false"` when there's at least one chip, maybe keep it hidden attribute if none (less clutter for screen reader).
    - Call `layoutRadialChips()` whenever:
    - Pinned panels change,
    - A pinned panel opens or closes (to hide/show chip).
    - Window resize (for reposition).
    - We also set up maybe a `window.addEventListener('resize', layoutRadialChips)` to reposition chips on viewport changes.

20. Initially, after state load, call it to render initial pinned chips (if any pinned exist). Also ensure if shell is overlay and user hasn't hidden it, these chips might be shown behind it; but since shell is pointer events auto and covers a portion, chips below it might be clickable if behind transparent areas. To avoid weirdness, maybe hide radial chips when shell overlay is visible, since you don't need them when directory is open.
    - Implement that: on `setSidebarMode('overlay'/'docked')`, set `menuRingEl.style.display = 'none'` (or aria-hidden true) to hide radial chips. On hide mode, `menuRingEl.style.display = 'block'` to show them.
    - Or simpler: If shell mode != hidden, hide radial menu.
    - This prevents UI overlap confusion.

21. Remove any outdated menu ring code:

    - The code snippet around L15845 in app.js shows `if (menuRingEl) menuRingEl.removeAttribute('aria-hidden');` [74] which implies it was hidden initially then shown. We will manage this ourselves; we can keep menuRing hidden until first needed.

22. **Implement Desk Bar UI:**

23. Insert HTML for desk bar: `<div id="deskBar" class="desk-bar"></div>` at bottom of body. Already in index.html we saw `<div id="floatingDockLayer" aria-live="polite"></div>` [75] which might have been intended for similar purpose.
    - We could reuse `#floatingDockLayer` as the container for icons (floatingDockLayer covers full screen though). But better, we insert a narrower `<div id="deskBar">` inside `floatingDockLayer` or as separate. The `floatingDockLayer` was pointer-events none full area; it might have been meant to catch panels moved to dock. We can possibly repurpose:
    - Actually, the code uses `floatingDockLayer` to append floating panels in dock mode (like when user pinned to left, they inserted a placeholder then moved panel into floatingDockLayer at pos).
    - Given we no longer dock panels to edges, `floatingDockLayer` usage can be changed.
    - E.g., we could use it as the fixed overlay for chips and bar (since it's already fixed full screen).
    - Or simply not use it and use our own. The simplest: Use `floatingDockLayer` as an empty container for any floating mini UI like deskBar and possibly radial too. It's fixed and pointer-events none by default, but we can override pointer-events on children.
    - It has `aria-live="polite"`, which could announce new items for accessibility. Good for our desk icons coming and going.
    - So, we can:
        - Append our `menuRing` and `deskBar` as children of `floatingDockLayer` (floatingDockLayer is currently empty).
        - Then ensure `floatingDockLayer` has pointer-events none globally, but we override for our UI: Actually easier: remove `pointer-events: none` from floatingDockLayer (or set it to auto), since we now want to interact with children. Or set none but each child container gets pointer-events auto (maybe simpler to just allow floatingDockLayer to handle events normally).

        - We'll check if any script sets floatingDockLayer.style pointer events none (we saw at [18†L13-L16] it does pointerEvents none).
        - It might be safe to remove that style.
    - We'll proceed using floatingDockLayer as parent for consistency with earlier code.

24. CSS:
    - `.desk-bar` style as described earlier. Also style `.desk-bar__icon` and any active states.

25. JS:
    - Maintain a list or map of open panels associated with desk icons.
    - Write function `updateDeskBar()` that:
    - Iterates through `state.ui.openPanels` (or a separate structure for currently open including minimized) and ensures each has an icon in deskBar.
        - If an icon for that panelId doesn't exist, create one:

```
let icon = document.createElement('button');
icon.className = 'desk-bar__icon';
icon.dataset.panelId = id;
icon.title = panelName;
icon.innerHTML = `<span class="desk-bar__icon-img">${iconSymbol}
</span>`;
deskBar.appendChild(icon);
```

          Attach click event to icon:

```
icon.addEventListener('click', () => onDeskIconClick(id));
```

        - If it exists, maybe update its appearance (like active state).
    - Remove icons for panels no longer open:
        - Loop through existing `.desk-bar__icon` elements, if any whose panelId is not in openPanels state, remove them.
    - Set active highlight:
        - Determine which panelId is currently focused (we can track last focused in state or a variable updated in bringToFront).
        - Add class `.desk-bar__icon--active` or `--open` to that icon, remove from others.
    - We call `updateDeskBar()` whenever:
        - A panel opens or is closed,
        - A panel is brought to front (to update active highlighting),
        - Possibly when a panel is minimized or restored (though that panel still considered open in our logic).
    - We can also integrate into `openPanel` (add icon) and `closePanel` (remove icon) directly rather than scanning each time. Might be simpler:
        - In openPanel, after making visible, call `addDeskIcon(panelId)`.
        - In closePanel, call `removeDeskIcon(panelId)`.
        - In bringToFront, call `highlightDeskIcon(panelId)`. This way we update incrementally.

- We will implement helper `addDeskIcon(id)` that creates and appends icon (if not exists), and `removeDeskIcon(id)` to remove it, and `highlightDeskIcon(id)` to mark active.
- `onDeskIconClick(panelId)`: logic:
- If the panel is currently not visible (hidden attribute true because minimized), then we call `openPanel(panelId)` to restore it.
- Else (panel is visible):
  - If it's not focused (meaning another panel is on top of it), then bring it to front (focus it).
  - If it is already focused and visible, we might optionally minimize it (but as per earlier decision, we won't on single click).
  - So effectively for visible but behind: focus it. For visible and front: no action (we could flash it or something, but not needed).
- We can track panel's hidden status by checking `panel.hasAttribute('hidden')` or a state variable.
- Probably easiest: retrieve panel element. If `panel.hidden == true` (means collapsed), do openPanel. If `panel.hidden == false`, do bringToFront.
- Also consider: if user uses desk bar icon to focus a panel that was behind others, we want to raise it. If user wants to minimize an open panel, they have to click its collapse in header (we won't do via icon to keep consistent UI control).

26. Ensure deskBar appears only when there is at least one open panel:

    - Could set `deskBar.style.display = 'none'` initially, then when first icon added, set display flex.
    - And hide when last removed.
    - Or simply leave it empty vs not in DOM. But for cleanliness, toggling display is good.
    - The `aria-live="polite"` on floatingDockLayer might announce new "button" added which screen reader will read label, that's good.

27. **Tie into Application State & Persistence:**

28. At app load, after retrieving saved state from localStorage:
    - Initialize `state.ui.pinnedPanels` (and others) from stored data or default.
    - Immediately reflect pinned by rendering radial menu (if shell hidden by default).
    - If `state.ui.shellMode` was persisted e.g. 'docked', call `setSidebarMode('docked')` to apply correct class and toggle text [76].
    - If any `state.ui.openPanels` persisted, for each call `openPanel(id)` but perhaps without animation and with a slight delay to avoid overwhelming. Possibly just open the last opened or none. This could be optional; maybe better to not auto-restore open panels on reload except maybe one main panel. The spec doesn't explicitly say to reopen last session's windows, but we assumed persistent open.
    - We can do it to be thorough: for each id in saved openPanels, call openPanel. This will create icons and show them. But if there were many, it could start cluttered. Perhaps only do it if user had pinned or something.
    - Given we persist pinned anyway, maybe auto-opening everything is not needed. Possibly skip restoring open by default (like an OS reboot typically doesn't reopen all your windows unless configured).

- I'll lean to *not* auto-open panels on load for simplicity, just persist pinned. The user can easily open pinned via radial.
- So, `state.ui.openPanels` can be used for runtime tracking but not necessarily persisted (we can decide not to persist openPanels list, or persist but not automatically act on it).
- We will persist last positions and such though so if user opens the panel again it could appear where left off.
- To keep it simpler: we won't auto restore open panels, except maybe certain always-on like Entry Door in some flows (but that one is fixed = true anyway and not in directory listing).

29. Update `persist()` or usage:

    - Ensure after we change pinnedPanels or toggles we call persist to save.
    - Possibly refine persist to not store ephemeral things (like openPanels if we choose not to restore them).
    - The dev journal indicates `persist()` saves to localStorage [77]. We must call it when appropriate:
    - After pin/unpin, after toggling shell mode, after closing maybe (to save positions).
    - And inside existing state toggles like field changes they already call persist.
    - Save panel positions: The current code's `panelDockState` presumably was persisted. If we keep using it or our own, when a panel moves (drag end event), update `panelDockState[id].x/y`.
    - There likely is already a drag handling that updates it (the code in `initDockablePanel` prepared placeholder for dragging).
    - We might adopt a simpler approach for now: not implement free drag persist fully (if not trivial). Or if we rely on `panelDockState`, it's already part of state and persisted. Actually, they fill `entry.x` and `y` in initDockablePanel [39] using either default or existing entry, and presumably update it on drag.
    - Possibly a function like `updatePanelPosition(panelId, x,y)` that sets state and maybe uses `persist()` eventually. This may have been partially done in old code, but let's see if not too complex to integrate:
        - We can attach to panel `onmousedown` and track `onmouseup` to compute final positions and call update state then persist.
    - If not done now, not critical; panels can reopen centered each time.

30. **Testing & Debugging:**

31. With all pieces in place, test common flows:
    - Pin and unpin from directory (shell overlay mode).
    - Search filter sort in directory.
    - Open panel from directory (shell overlay): does it open, and does desk icon appear? Does radial chip hide if it was pinned?
    - Close panel via header X: does it disappear and remove desk icon and show radial chip back if pinned?
    - Collapse panel via header _: if pinned, does panel hide and chip remain? If not pinned, does icon remain in desk bar (since it's still open but hidden)? Actually if not pinned and collapsed, per our plan, we should move it to desk bar (but it was already open so already in desk bar; now just remains with panel hidden, which is fine).

- Focus via header or naturally: open two panels, click focus on one’s header, ensure it's in front.
- Use desk bar icons to switch between open panels: open 2-3, click icons to bring forward.
- Hide shell (shell mode Hidden): radial chips should appear for pinned, desk bar still accessible for open ones, etc. Show shell again (shell overlay or docked) to see it updates.
- Try pinned limit: pin more than 5, ensure 6th doesn’t pin or UI stops it.
- Try filter categories off for some, see list updates.
- The "Hide button auto-hides when shell empty": if you filter out all panels or on initial maybe no panel scenario (contrived), does toggle hide? We may simulate by temporarily making `state.ui.filters` all false, shell list empty, then `#sidebarShell.dataset.empty` becomes true and CSS hides toggle. Then re-enable and see it come back.
- Check persistence by reloading: pinned remain pinned (radial chips show), shell mode persists (if last was docked, it loads docked), filter/sort maybe persist (if we saved them).

32. Debug any JS console errors or style issues. Adjust accordingly.

33. **Clean Up Deprecated Code:**

34. Remove any functions and variables that are no longer needed:

    - `panelDockState` may be partially obsolete. If we no longer dock panels left/right, we may reduce it to just store positions or minimization state. Or we can repurpose the 'mode' field for our 'circle'/'desktop' states (we saw code checking `'membrane-panel--circular'` classes [7]).
    - The `initDockablePanel` function in app.js (with slider and circleBtn etc. [40] [47]) can be heavily simplified or removed:
    - We don’t need to create the pin left/float/right slider UI – scrap that part.
    - We don’t need unbundle left/right or off (off = close, but we have our close).
    - We *do* want to ensure each panel is positioned properly at open, which initDockablePanel was doing (calculating default x,y center and storing in panelDockState).
    - We can keep a trimmed down version: when panel is opened first time, if not in state, set its x,y to center. We can do that in openPanel logic instead.
    - So likely, remove calls to initDockablePanel for each panel and replace with our openPanel.
    - There might be code calling initDockablePanel when building UI (maybe after creating each panel element). Search usage of initDockablePanel in code: We saw it being called for presumably each panel in a loop or individually in some init routine (maybe scanning .membrane-panel class and calling it). If so, remove those or stub them out. Instead, might just rely on openPanel to set default position.
    - Remove `cycleSidebarMode()` if not needed (but we still use it for toggle).
    - Confirm `sidebarShellToggle` event is still appropriate: yes, it cycles Hidden/Overlay/Docked. Possibly we might want a direct hide/unhide separate from cycling, but spec specifically said 3 modes, so it's fine.
    - Remove ghost aside HTML (`sidebarRight`) from index if we haven't, and related CSS `.sidebar`.
    - Possibly remove dev/test code in HTML or any commented out parts after verifying everything works.

35. **Finalize Styling & Micro-interactions:**

    - Add finishing touches:
    - Panel glass/blur effect: ensure panels have semi-transparent background with backdrop-filter. Possibly define `--panel-bg: rgba(0,0,0,0.5)` for a nice dark translucent, and apply to `.membrane-panel` via CSS: `background: var(--panel-bg); backdrop-filter: blur(8px); border: 1px solid var(--panel-border); border-radius: 8px;`.
    - Inset scrollbars: check panel content scrollbars, style similarly to sidebar if needed (e.g., `.membrane-panel::-webkit-scrollbar` etc).
    - Smooth scrolling: add `scroll-behavior: smooth;` to the shell body and maybe panel bodies so any programmatic scrolls are smooth.
    - Hover transitions: e.g., the radial chips we did, maybe also on header control buttons (change color on hover).
    - Active focus outlines: ensure keyboard users can see focus on icons (we added outline to desk icons).
    - Pin icon toggle: if using text "Pin"/"Unpin", we should update the button text on toggle. Or if using an icon, possibly rotate or change icon to indicate pinned state. We'll implement at least text swap or a visual change (like a filled pin vs empty – if using emoji, maybe vs).
    - Possibly incorporate a lightweight icon library or SVGs for a more polished look for the control icons. But since we aim for vanilla, maybe skip if not already present.
    - Test on different screen sizes (responsive): on a smaller window, the dock might overflow – our scroll solves it, radial might bunch up – but with <=5, still okay on small.
    - Ensure no content is out of place in docked mode (when shell is docked left, does canvas behind adjust? We currently overlay it, so donut might be partly obscured. That might be okay).
    - If we wanted, we could shrink canvas width when docked to give space (like shift camera), but not planned.
    - Transparent hide button: maybe style the sidebarShellToggle to be a ghost icon (like an eye) when hidden mode is engaged. But can skip detailed design of that.

36. **Documentation & Comments:**

    - Add comments in code for major sections to explain the new approach for future devs.
    - Possibly update `dev_journal.md` with an entry describing this overhaul (though that's beyond what user asked, but might mention in collab note that we should record it, as they have entries logging changes).
    - Confirm that all design requirements are met or note if any divergence.

Completing these steps will replace the old membrane navigation system with the new **Modular Membrane UI**. We will proceed incrementally, testing each stage thoroughly (especially the delicate interactions between shell, radial, and desk components). This approach minimizes regression risk and aligns with the user's redesign spec. The resulting codebase will be cleaner (removing ghost/docking cruft) and ready for further enhancements.

Extra “Cool” Ideas (Innovative Additions)

Beyond the core requirements, here are some additional ideas to enhance the experience, prioritized from most feasible/impactful to more experimental:

1. **Animated Panel Transitions:** Add smooth animations when panels open, collapse, or close. For example, when collapsing a panel, animate it shrinking into its radial chip or desk icon (a quick scale-down and fade-out towards the chip’s position). This gives a clear visual cue of where it went. Similarly, opening a panel from a chip could animate the chip expanding into the panel. These micro-interactions reinforce the spatial metaphor (panel <-> chip relationship) and polish the UI.

2. **Workspace Presets / Layout Save & Recall:** Allow users to save the current arrangement of open panels and donut settings as a **preset** (workspace). This matches the idea of “Intention for this passage” in Entry Door – each intention could map to a preset layout of panels relevant for that mode. For example, a “Focus” preset could auto-open the Brain Waves and Field Deck panels in a certain arrangement, whereas a “Visualize” preset opens the Everything Chalice and LoL panels, etc. We can implement this by serializing `state.ui.openPanels` and panel positions into a preset object, and providing a UI (maybe in the Entry Door panel or a top toolbar) to switch presets. This makes it one-click to configure the UI for a particular task or mood.

3. **Soft Parallax Effects:** Introduce subtle parallax movement for UI elements relative to the donut. As the user moves the mouse or as the camera orbits slightly, the radial chips and panels could shift gently, creating depth. For instance, the radial menu could rotate or wobble slightly in opposite direction to donut rotation (small offset on angles) to emphasize 3D space. Or panels could have a slight 3D tilt (using CSS `transform: perspective()` and rotate a couple degrees on hover or with device orientation), as if they are physical floating screens around the donut. This adds a dynamic, immersive feel without altering functionality.

4. **Gesture and Touch Support:** Though desktop-first, we can add enhancements for touch and trackpad:

5. Implement a **swipe gesture** to quickly hide or reveal the shell: e.g., a two-finger swipe from left edge could dock or hide the sidebar. Or dragging the sidebar toggle button could cycle modes.
6. For panels, allow a swipe down gesture on a panel header to collapse it (minimize), and maybe a pinch on a panel to scale it (if resizing implemented in future).
7. On touch devices (or if using something like a Leap Motion or VR controllers), the radial menu could support dragging a finger around the donut to highlight chips (like a virtual rotary selector).
8. Use a small library (like Hammer.js) for complex gesture detection if needed (lightweight and can handle swipes/pinches easily).

9. Also, enable long-press on radial chips to show labels (since hover isn’t available on touch).

10. **Enhanced Visual Feedback (Hover Glow & Sound):**

11. When hovering over radial chips or panel control buttons, add a **glow or particle effect**. For example, a faint circular ripple emanating from a chip on hover, echoing the donut’s aesthetic.

12. Use the accent color for these micro-highlights. This can be done with CSS `box-shadow` pulses or canvas if feeling fancy.
13. Tie in subtle **sound effects**: a soft "pop" or sci-fi click when opening/closing panels, a tiny tick sound when pinning or a whoosh when toggling the shell. Keep volume low and make it optional, but this auditory feedback can enhance the futuristic feel.

14. If EEG is connected (Neurosity Crown), perhaps use brainwave events to trigger slight UI feedback (e.g., a focus event could make the active panel glow more).

15. **Intelligent Auto-Layout:** Implement an optional "smart arrange" function where the system automatically arranges open panels to avoid overlap. For instance, a button that when clicked, tiles all open panels around the donut (maybe in a circular fashion or grid). This could quickly declutter if the user has many open. We could use simple heuristics: e.g., if 3 panels open, place them at 120° intervals around center, etc. (This is complex to do perfectly but even a rough distribution might help.)

16. **Minimap or Overview Mode:** Provide a bird's-eye view of all panels in a miniature form. For example, a special key (or pinch gesture) could zoom out the UI so all open panels shrink and gather, showing an overview of what's open (like Mission Control on macOS). The user could then tap a panel to focus it or close it quickly from that overview. This leverages the 3D canvas – perhaps the panels could even tilt in 3D around the donut in this mode, which would look very "sci-fi control center".

17. **Contextual Panel Suggestions:** When the user performs certain actions, suggest relevant panels via a brief highlight of their chip or an alert. For example, if the user connects the Crown (EEG), automatically pulse the Brain Waves chip to suggest opening it. If they change a donut geometry parameter drastically, highlight the Field Deck panel. These gentle cues can help users discover features at appropriate times.

18. **Improved Iconography:** Replace text labels or basic emojis with a cohesive icon set for all panels and controls. Possibly design custom minimalist icons that match the donut aesthetic (line-art, neon glow style). E.g., a brainwave icon, a torus icon, a pin icon, etc. This would make the UI more visually intuitive and professional. We can integrate an SVG sprite or icon font. (This is lower priority in code, but high in UI polish.)

19. **Dynamic Theme or Mode based on "Intention":** Tie the UI's look and feel to the current Intention (Entry Door selection). For instance, in "Focus" mode, use a certain color scheme (maybe dim and cool colors), in "Create" mode use vibrant colors and maybe more bouncy animations, etc. We can define CSS variables for each intention theme and switch them when intention changes. Also, possibly auto-open a certain set of panels per intention (which overlaps with workspace presets idea). This makes the entire interface adapt contextually to the user's goal.

Each of these ideas extends the system in a way that complements the **3D donut aesthetic and interactive vibe**. They are modular enhancements: we can implement them gradually. The first few (animations, workspace presets, parallax) are highly recommended as they add clear value and can be done with reasonable effort (using CSS transitions, leveraging existing state). The later ideas (overview mode, contextual suggestions) are more experimental but could differentiate the experience significantly.

Importantly, all are optional and won't break the core functionality if omitted, so we can prioritize based on user feedback and development resources.

Open Questions

Finally, a few open questions and decisions need clarification as we implement the redesign:

- **Should we allow the user to choose between Circle mode and Desk mode, or use both concurrently as in our plan?** The spec lists them separately, implying perhaps a toggle (maybe a "bundle" all to desk vs radial layout). Our implementation uses both (radial for favorites, desk for open tasks). We should confirm with the design team if they envisioned a single mode active at a time (e.g., a user setting: "Minimize panels to: [Radial Ring / Desktop Bar]"). If so, we'll adjust by providing that option and mutually exclusive behavior. If concurrent is acceptable (which we assumed to cover dense vs few cases), we'll proceed with that hybrid approach.

- **What is the exact categorization of each membrane panel into Signals/Geometry/Visual/Meta?** We have inferred some categories (e.g., Crown & Brain Waves as Signals, Donut builder as Geometry, etc.), but a definitive mapping from the product owner would ensure the filter is accurate. For example, does "Meta" include Entry Door and Membrane Directory (which anyway are fixed/hidden)? Are Intention/Labels considered Meta or Visual? We can easily adjust the `data-membrane-category` tags once we get the list.

- **How to handle panel processes when panels are not visible?** Specifically, should collapsing or closing a panel stop its ongoing activity? For instance, if Brain Waves panel is streaming data and the user collapses it, do we keep streaming (so that if reopened the data is continuous)? Or pause it to save resources? Similarly with Neurosity Crown connection or Field Deck controlling donut parameters – some panels affect the donut even when UI is closed. We need a policy:

- Option 1: Keep processes running until panel is explicitly turned off or closed (UI just hides). This means e.g. you can collapse Brain Waves but it keeps logging in background.

- Option 2: Tie process to panel visibility: closing panel turns off that feature. This decision affects user experience (maybe they want EEG always on regardless of panel). We should clarify the expected behavior per feature. We might implement defaults (e.g., keep Crown link alive unless user disconnects explicitly, even if UI closed, since they might collapse UI to focus on donut visualization).

- **Do we need to support dual-side docking (left & right) in any form?** The design removed the ghost right sidebar, but the code still had provisions for docking to right. Should we completely remove the ability to pin a panel to a side (embedded in interface) now? We assumed yes (panels always float or minimize, no sticking into a sidebar). Just to confirm: if a user wanted, say, to always see Brain Waves panel on the right side, would we disallow that now? Perhaps the new design intentionally avoids fixed side panels in favor of overlays. We should verify with stakeholders if persistent side panels are a use-case to preserve. If needed, we could allow docking one or two panels to screen edges as an advanced feature (maybe via right-click on collapse to "dock left/right"). For now, we dropped it, but it's worth confirming if that's acceptable.

- **Will there be any integration with XR (AR/VR) interactions for the UI?** Dev notes mention XR spatial UI [54]. If the donut is used in VR mode, the 2D HUD elements (shell, chips, dock) might need adjustment (like rendering in VR space or alternative UI). This might be outside current scope, but planning ahead: possibly have an XR mode where the radial menu becomes actual 3D buttons around the donut, etc. For now, we treat UI as 2D overlay only, but if XR support is planned, we might architect things so that the panel system could be switched off in VR and replaced with spatial menus, etc.

- **Performance considerations with many panels:** The system should handle, say, a dozen panels open/minimized without lag, but we should confirm if any part of the code (like continually calling renderMembraneDirectory filtering all .membrane-panel nodes [78]) is okay performance-wise. It's likely fine (dozens of elements), but if we had a scenario of a user spawning many custom panels (the + Custom Membrane button?), we should keep an eye on performance. We might ask: what is the upper bound of membrane panels a user could have listed? If it's large (50+), we might need to optimize the directory rendering (virtualize list or such) or at least ensure search is efficient. This is something to watch as usage patterns emerge.

By addressing these questions with the design/development team, we can fine-tune the implementation. Once resolved, we'll update the plan accordingly (for instance, enabling a global "minimize to desk vs ring" setting if needed, or adjusting the filter category logic). Overall, these clarifications will help ensure the final UI matches both the designers' intent and user expectations.

---

[1] [4] [5] [6] [11] [21] [37] [49] [52] [61] [62] [64] [65] light.css
file://file_00000000a51471f48b92f0cd1a479f1f

[2] [7] [8] [9] [12] [13] [14] [15] [16] [17] [18] [19] [23] [24] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [38] [39] [40] [41] [42] [46] [47] [48] [50] [51] [66] [67] [70] [71] [72] [74] [76] [77] [78] app.js
file://file_0000000022b072469509d9e04eaaaa91

[3] [20] [22] [25] [43] [44] [45] [53] [59] [60] [68] [69] [73] [75] index.html
file://file_00000000cd047246861ac02787847677

[10] [54] [55] [56] [57] [58] [63] dev_journal.md
file://file_000000007a4471f4992375ed4fef4429