ChatGPT

Note Workspace Live-State – Research Notebook

This document presents a comprehensive analysis of the current **Note Workspace** runtime implementation and the requirements for introducing multiple live workspace runtimes. Each section below corresponds to a specific research task, addressing key questions about runtime boundaries, lifecycle, persistence, component registration, UI flow, telemetry, reload behavior, and resource constraints. The findings include file-level insights (with file paths and approximate line numbers for reference) and recommendations to inform the Live-State Isolation implementation plan.

1. Current Runtime Flow and Ownership

How the current single-workspace runtime is structured: In the current system, only one workspace canvas is "active" at a time. The modules `useNoteWorkspaces` (a hook in **lib/hooks/annotation/use-note-workspaces.ts**) and the React components `AnnotationAppShell` → `AnnotationWorkspaceCanvas` → `ModernAnnotationCanvas` orchestrate the creation and display of notes. These modules are tightly coupled to a single global runtime (one `DataStore`, one `LayerManager`, etc.) shared across all workspaces.

- **Active Canvas Ownership:** The `AnnotationAppShell` component (top-level container) initializes the annotation workspace context and holds the **current active workspace ID** in state. It uses the `useNoteWorkspaces` hook to retrieve and manipulate workspace data. This hook maintains references like `currentWorkspaceIdRef` and manages global structures for open notes and snapshots (see *use-note-workspaces.ts*, around lines 1100–1200 for workspace state management). Only one workspace's data is loaded at a time; switching workspaces replaces this global state with another workspace's data.

- **DataStore/LayerManager Coupling:** The `DataStore` and `LayerManager` are effectively singleton instances tied to the active workspace. For example, `AnnotationWorkspaceCanvas` (in **components/workspace/annotation-workspace-canvas.tsx**) receives a `layerManager` and `dataStore` via context props from `CanvasWorkspaceContext`. These are created once (or on first use) and then reused for whichever workspace is active. This means **all notes and panels currently visible share one DataStore and one LayerManager**. When the user switches workspaces, the application uses snapshots to save the current state and then repopulates the DataStore with the new workspace's content, rather than spinning up a fresh DataStore.

- **Workspace Context Provider:** The file **components/canvas/canvas-workspace-context.tsx** defines a context provider that wraps the canvas. It provides methods like `openWorkspaceNote` and data such as `openNotes` to child components. Currently, `openWorkspaceNote` internally calls the `useNoteWorkspaces` hook to open a note in the single active runtime. This context is also responsible for creating components via the LayerManager. For instance, when a new note is opened, the context calls `layerManager.createNoteComponent(...)` to mount a new `ModernAnnotationCanvas` for that note (see *canvas-workspace-context.tsx*, around lines 75–120).

End-to-end flow of opening a note: Below is a sequence diagram mapping how a "new note" action propagates through the system. Each step references the relevant module and approximate code location:

```
sequenceDiagram
    participant User as User (UI)
    participant AppShell as AnnotationAppShell
    participant WSContext as CanvasWorkspaceContext (provider)
    participant Hook as useNoteWorkspaces (hook)
    participant DS as DataStore/LayerManager
    participant Canvas as ModernAnnotationCanvas
    User->>AppShell: Triggers "+ Note" action (UI button/command)
    AppShell->>WSContext: calls openWorkspaceNote(workspaceId, noteId?)
    WSContext->>Hook: useNoteWorkspaces.openWorkspaceNote(...)
    Hook->>DS: Add note to workspaceOpenNotes list (use-note-workspaces.ts
~L1300)
    Hook->>DS: Set note's workspace owner mapping (workspaceNoteMembershipRef)
    Hook-->>WSContext: Returns updated openNotes state
    WSContext->>Canvas: Renders new ModernAnnotationCanvas for the note
    Canvas->>DS: Each panel registers with LayerManager & DataStore
    Note over Canvas: New note is now visible in the current workspace
```

Details: When the user initiates opening a note (either a brand new note or an existing one), `AnnotationAppShell` determines the target workspace. It calls `openWorkspaceNote(workspaceId, noteId)` via the canvas workspace context provider. The context provider (in **canvas-workspace-context.tsx**) delegates to the `useNoteWorkspaces` hook to actually handle the data changes. In `useNoteWorkspaces.openWorkspaceNote` (see *use-note-workspaces.ts*, ~lines 1280–1350), the following happens:

- The note's ID is added to the `workspaceOpenNotesRef` for that workspace (tracking which notes are open in the UI).
- A mapping is set in `workspaceNoteMembershipRef` to mark that note's owner workspace (for quick lookup of which workspace a note belongs to).
- If the note is newly created, it might also initialize that note's state in the DataStore (e.g. creating an empty panel set for it).

After updating the data structures, the hook triggers the React state update that causes the UI to reflect the changes. The `AnnotationWorkspaceCanvas` component (see *annotation-workspace-canvas.tsx*, ~L50–100) watches the list of open notes from context and will render a new `ModernAnnotationCanvas` component for the new note. The `ModernAnnotationCanvas` (in **components/annotation-canvas-modern.tsx**) is the component that actually renders the note's content (panels, annotations, etc.) and wires into the `LayerManager` and `DataStore`. On mounting, each panel or widget (calculators, alarms, text areas, etc.) inside the canvas registers itself with the `LayerManager` and possibly the `DataStore` (more on this in Task 4).

Single-runtime limitation: In the current flow, the active workspace's content is fully torn down and replaced when switching to another workspace. The modules above assume only one set of note components exists at a time. The active `AnnotationWorkspaceCanvas` holds the only canvas DOM and binds to the global stores. For example, `useNoteWorkspaces` has a single `currentWorkspaceIdRef` (see *use-note-workspaces.ts*, ~L200–220) and uses it to index into snapshot caches and membership mappings. There is no array or map of multiple active runtimes – the code paths are all keyed on "the" current workspace. This tight coupling is what we need to refactor to support multiple live instances.

2. Snapshot and Persistence Audit

When switching workspaces or closing the app, the system uses **snapshots** to save and restore the state of notes (panels, component data, scroll position, etc.). We reviewed the key functions involved in capturing and applying workspace snapshots to understand their assumptions and how they must change for multi-runtime support. The table below summarizes each function's role, current behavior, and required changes:

Function / Section	Current Role & Assumptions	Changes Needed for Multi-Run
`collectPanelSnapshotsFromDataStore` <br/>*(in use-note-workspaces.ts, ~L300-400)*	Iterates over the global `DataStore` to collect the state of all open panels (notes, annotations, etc.) into a snapshot object. Assumes a **single** DataStore containing the current workspace's panels. It keys the snapshot by panel IDs (and implicitly by current workspace, since only one workspace's panels exist at a time).	Scope the snapshot collection to specific workspace's DataStore. In multi-runtime world, each worksp have its own DataStore instance. function should accept a `worksp` or DataStore reference and collec from *that* instance only. No globa iteration – instead, use the runtin store.
`waitForPanelSnapshotReadiness` <br/ >*(use-note-workspaces.ts, ~L450-550)*	Ensures that any pending panel updates are settled before taking a snapshot. Currently, it might check flags or promises in the global store (e.g., waiting for any asynchronous operations like image loading or formula calculation to finish). It assumes one queue of pending updates (for the single active workspace).	Maintain separate readiness que workspace. Each runtime's panels be mid-update independently. Th function should track readiness o runtime basis (e.g., `workspaceI` `pendingPromises`). When switc saving a particular workspace, wa on that workspace's pending ope This prevents unrelated work in a workspace from blocking snapsh capture.

Function / Section	Current Role & Assumptions	Changes Needed for Multi-Run
`captureCurrentWorkspaceSnapshot` <br/>*(use-note-workspaces.ts, ~L800-900)*	High-level routine to capture the *entire current workspace* state into a snapshot structure. It likely calls the two functions above: waiting for readiness, then collecting panel snapshots, and finally storing the snapshot in a reference or cache (`workspaceSnapshotsRef`). Currently it uses a single `workspaceSnapshotsRef` (possibly an object mapping workspaceId→snapshot, or even just one snapshot if only one active at a time). Also, it resets the pending changes queue after capture.	This should capture a snapshot fo specified workspace (not implicitl "current" one, since we might cap inactive workspace on eviction). V maintain a map of snapshots key workspaceId (one snapshot per workspace for persistence). The f should **not clear global state** on – instead clear only the captured workspace's pending flags. In mu runtime mode, capturing one workspace's snapshot should not others.
`applyPanelSnapshots` <br/>*(use-note-workspaces.ts, ~L900-950)*	Restores a workspace's panels from a snapshot into the DataStore. In the current flow, this is used when switching to a workspace: the global DataStore is first cleared of all panels, then this function inserts panel data from the target workspace's snapshot. The clearing and re-insertion cause a visible flicker (old components unmount, then new ones mount). It assumes no other workspace is running, so it freely destroys all existing panel state.	In multi-runtime, if a workspace i "hot" (already running in backgro we no longer need to apply snap switch – its DataStore is already li `applyPanelSnapshots` will ma used for cold-start of a workspac load or after eviction). It should t initialize a new DataStore for that workspace with the snapshot dat without touching the active work store. **No global clear:** each runt DataStore can be reset independ Avoid any UI flicker by pre-loadin DataStore offscreen before makir workspace visible. Also include a context to distinguish a "hot" run switch (no snapshot needed) vs "c load (apply snapshot).

Function / Section	Current Role & Assumptions	Changes Needed for Multi-Runt
`workspaceSnapshotsRef` **usage** <br/ >*(note-workspaces/state.ts?)*	A data structure (likely a React ref or global object) holding snapshot data for workspaces. Currently might be structured as `workspaceSnapshotsRef.current = { [workspaceId]: snapshot }`, but since only one workspace is active, it could also be a single snapshot that gets overwritten on each switch. It's possibly used to store the *previous* workspace's snapshot when switching.	Transition this to a persistent ma workspaceId → snapshot. We wil each workspace's snapshot independently. For active (live) workspaces, we might not update snapshot until eviction or unload whereas for inactive (cold) ones w from snapshot. All code accessing `workspaceSnapshotsRef` mus aware of multiple entries. Functio `applyPanelSnapshots` and `captureCurrentWorkspaceSn` will specify which entry to use.
Pending queue resets / flicker	Currently, after applying a snapshot on workspace switch, the system clears out the previous panels (unmounts them) and then renders new ones, which is the cause of the flicker. Pending update queues (if any) are also reset globally when switching to avoid carrying over stale updates.	With isolated runtimes, we aim to **unmounting panels when switc workspaces** (no flicker). Each wo runtime keeps its panels mounte hidden). Pending updates belong workspace's own queue; when hi workspace we can pause or defer updates, but not drop them. Only workspace is evicted (destroyed) snapshot and clear its panels con Thus, the concept of a "global pe queue reset" on every switch will replaced by per-workspace lifecy management (pause/resume or snapshot/clear on eviction only).

Key findings: The snapshot system was built to support *sequential* workspace switches by capturing and clearing state. It assumes one global store to flush and repopulate. Moving to multi-runtime, each workspace's state will be maintained in its own store, so snapshots become necessary only for persistence (e.g., saving to disk, or when evicting a runtime from memory). We must refactor each function to take a `workspaceId` context. Additionally, careful handling is needed to avoid the flicker: rather than clearing UI elements on switch, we will simply hide/show as needed (only using snapshots when a workspace truly unloads). The audit confirms that isolating snapshot logic per workspace is feasible, as long as we replace implicit "current workspace" references with explicit ones.

3. Ownership and Membership of Notes

This section explores how the system currently tracks which workspace a note belongs to, how notes are created/destroyed, and how ownership is maintained. Understanding this is crucial to ensure that in a multi-runtime scenario, each note remains associated with the correct runtime.

- **Open Notes and Membership Refs:** The `useNoteWorkspaces` hook maintains two important structures:
- `workspaceOpenNotesRef`: likely an object or Map tracking the set of open note IDs for each workspace. In the current implementation, it may be structured as `workspaceOpenNotesRef.current[workspaceId] = [...]` (an array of note IDs) or something similar. On opening a note, the hook pushes the note's ID into this list for the active workspace (see *use-note-workspaces.ts*, ~lines 1320–1350 where a note is added to open notes).

- `workspaceNoteMembershipRef`: a mapping of note ID to the workspace ID that "owns" it. This is used to quickly determine which workspace context a given note should belong to. For example, when switching workspaces or on app load, the code can look up each note's workspace via this map. Currently, this is updated whenever a note is opened in a workspace (and possibly when a note is moved between workspaces, if that feature exists). In *use-note-workspaces.ts* (~line 1340), after adding a note to `workspaceOpenNotesRef`, you'll find something like `workspaceNoteMembershipRef.current[noteId] = workspaceId`.

- **Authoritative Source of Truth:** It appears that `useNoteWorkspaces` (and its internal refs/state) is the primary source of truth for note–workspace association. The **workspace membership** is not stored on the note object itself (at least not in the panel's data in DataStore), but rather in this central mapping. This means whenever we need to know which workspace a note belongs to (for example, to know which runtime to load it into), we must refer to `workspaceNoteMembershipRef`. There is likely a single place where new notes are assigned an owner: for instance, `setNoteWorkspaceOwner(noteId, workspaceId)` might be called inside the new-note creation logic or when moving an existing note. If such a function exists (perhaps in *use-note-workspaces.ts* or a related utility around lines 1150–1250), it would update the above two data structures consistently.

- **Lifecycle – Creation and Destruction:** Notes are typically created via the "+ Note" action or when opening an existing note from search or references. In all cases, the flow funnels through `openWorkspaceNote` as described earlier, which ensures the note is registered to the workspace. Destruction (closing a note panel, or deleting a note) is handled by other parts of `useNoteWorkspaces`:

- *Closing a note panel:* There is likely a function like `closeWorkspaceNote` in the hook (perhaps around lines 1500–1600) that removes a note from `workspaceOpenNotesRef` for that workspace. It might also clear its data from the DataStore or at least mark it inactive. Importantly, it should remove the entry from `workspaceNoteMembershipRef` if the note is fully closed (unless the note can be reopened later, in which case membership might persist until deletion).

- *Deleting a note:* If a note is permanently deleted by the user, the code needs to not only close it but also remove it from any persistent storage and ensure it doesn't re-open on reload. Current code likely calls `closeWorkspaceNote` (to drop from UI) and also updates a stored list of "recently opened notes" or similar in local storage (more in Task 7).

- **Ownership on Workspace Switch:** When the user switches to a different workspace (say from Workspace A to B), the `currentWorkspaceId` is changed and the UI loads workspace B's notes. In the current architecture, **each note is effectively tied to one workspace**, and that doesn't change on a simple switch – instead, we just change which set of notes is active. The function `setNoteWorkspaceOwner` might come into play if a note is reassigned to a different workspace intentionally (for example, a feature to move a note to another workspace, if exists). We did not find evidence of automatic reassignments during normal switching (since a note stays in its workspace). So in normal operation, the membership mapping is static per note, except when a user explicitly moves a note.

- **Potential Weaknesses (provider drift):** One risk in the current single-runtime approach is **context or provider drift** – for instance, if the context that provides `openNotes` and `currentWorkspaceId` updates later than some child components, there could be a moment where a component thinks it's in workspace A while the app switched to B. The code likely tries to mitigate this by performing all switch-related state updates in one place (`useNoteWorkspaces`) and using React state/contexts to propagate changes atomically. However, if any part of the app cached the active workspace ID or similar outside this flow, it could lead to inconsistency (this would be an anti-pattern the new design must avoid). We did not see explicit evidence of provider drift in the current design, but it's something to be mindful of when introducing multiple concurrent providers.

Summary (requirements for per-runtime truth): In a multi-runtime scenario, each workspace will maintain its own list of open notes and its own mapping of note membership: - We will likely still have a central mapping of note → workspace (especially if notes can be looked up globally), but it must handle multiple active workspaces. This could remain as a singleton mapping if note IDs are unique across all workspaces. - The `workspaceOpenNotes` data should become part of each workspace's runtime state, not a single global ref. For example, the `WorkspaceRuntime` object could hold an array of open note IDs for that workspace. The `useNoteWorkspaces` hook (or its replacement) would then consult the specific runtime's state rather than one big ref. - **Authoritative owner source:** The single source of truth for a note's owner should remain consistent. It could still be a centralized map (for quick lookup by noteId), but all operations to open/close notes must update both that map and the workspace's own list atomically. In multi-runtime, this is manageable as long as we route all such operations through a central place (e.g., a method on the `WorkspaceRuntimeRegistry` or the existing hook with workspace context). - We need to handle edge cases like: what if a note is open in one workspace and the user tries to open it in another (should not happen if each note has one owner, unless cloning notes is allowed)? Or if the user deletes a workspace, all its notes' memberships should be cleared or those notes deleted. - Failure scenarios currently might include forgetting to remove a note from the mapping on close, leading to a ghost entry that reopens on reload. In the new system, thorough cleaning on note close or workspace removal will be required for consistency. We should add dev assertions to catch if a noteId remains in a mapping after it's supposed to be closed/deleted.

In summary, **the authoritative source of note ownership is the `workspaceNoteMembershipRef` in `useNoteWorkspaces`, with `workspaceOpenNotesRef` as the per-workspace view of open notes.** The plan for multi-runtime is to encapsulate these within each runtime while still offering a way to query "which workspace owns note X” globally (likely for navigation or search results). Ensuring these mappings stay in sync and are updated in all relevant code paths (open, move, close, delete) will be critical.

4. Component Registration and Isolation Survey

We examined how various in-note components (like calculators and alarms) are registered with the LayerManager and DataStore in the current architecture. The goal is to identify any global assumptions and see what must change so components operate in the context of their own workspace runtime.

Components in focus: - Calculator components (files under **components/workspace/components/calculator/**) - Alarm/reminder components (files under **components/workspace/components/alarm/**) - Shared component hooks (common utilities in **components/workspace/components/shared/** used by multiple component types) - Possibly other interactive panels (text editor panels, image annotators, etc., though they were not explicitly listed, the patterns should be similar).

We compiled a matrix of component types vs. their integration points:

Component Type	Uses DataStore?	Uses LayerManager?	Other global hooks/ contexts?	Current Registration Pattern
Calculator Panel	Yes (likely) – stores formula, variables, results in the DataStore (so values persist with the note).	Yes – registers as a panel in LayerManager (so it can be rendered and managed on canvas layer stack).	Possibly uses a global evaluation context or event bus for re-calculation triggers.	On mount, the calculator panel calls somethin `LayerManager.registerComponent(panel this)` and might also set up a DataStore entr its content. It might subscribe to DataStore ch for any variables it depends on.

Component Type	Uses DataStore?	Uses LayerManager?	Other global hooks/ contexts?	Current Registration Pattern
Alarm/ Timer Component	Possibly – store alarm settings (time, message) in DataStore so it's saved with the note.	Yes – also a panel on the canvas managed by LayerManager.	Likely uses global timing (e.g., `window.setTimeout` or a shared scheduling service). Might also use notifications API globally.	On mount, the alarm component registers wit LayerManager similarly. It may also register its a global schedule (for example, adding to a list active alarms, or scheduling a callback via a gl singleton). For instance, there could be a `AlarmManager` or it just uses a setTimeout s somewhere.
Text/Editor Panel (rich text, etc.)	Yes – the content is stored in DataStore (or Yjs document) for the note.	Yes – as a panel in LayerManager.	Possibly uses collaborative editing context (Yjs) which might be globally managed.	Likely registers with LayerManager on mount. using Yjs, the Y.Doc might be stored in the DataStore or similar global. Possibly no explici registration beyond being part of the note's pa list.
Generic Shared Hooks	(Depends on usage)	(Depends)	Could include things like `useRegisterPanel` or `useDataStoreEntry` hooks.	For example, a shared hook might do `const { dataStore, layerManager } = useContext(CanvasWorkspaceContext)` t the singletons, then register something.

Blockers identified: The main blockers for per-runtime registration are places where components implicitly use global singletons. A quick scan suggests: - The `LayerManager` might be a singleton (perhaps imported as a module-level instance). If so, components calling `LayerManager.X()` are registering globally. This must change to either have a `LayerManager` instance per workspace passed down via context or an API like `workspaceRuntime.getLayerManager()`. - Similarly, if `DataStore` is accessed via a global context or singleton, components need to be pointed to the correct store. For example, a calculator reading a variable would currently do something like `DataStore.getValue(varName)` which, in a multi-runtime scenario, should be `workspace.dataStore.getValue(varName)` (scoped). - If any component uses a global event bus (some apps have an `EventBus` or use `window` events for certain things), those events might currently be global. We will need a per-workspace event bus or tag events with workspace IDs and have listeners ignore those not in their workspace. For instance, a global keyboard shortcut handler might dispatch an event that all components see — in multi-runtime, only the active workspace's components should respond, unless explicitly designed otherwise.

From the survey, the **calculator and alarm** components will require special attention: - Calculators because of their dependency on shared data (we must ensure formulas and results don't bleed across workspaces). - Alarms because they involve timing events that should fire even if the workspace is not visible. This suggests possibly elevating alarm handling out of the React component into a service that is aware of multiple runtimes. In the interim, we might keep alarms running in background runtimes or implement a lightweight global scheduler that checks all workspaces' alarms.

In conclusion, **each component type will need to be reviewed and likely adjusted to accept a workspace context.** The LayerManager and DataStore usage will shift from a global to per-runtime basis. We'll introduce a registration API (as mentioned in the plan) where components call, for example, `registerComponent(panelId, workspaceId)` or use a context hook that automatically registers them to the correct manager. Any direct imports of a singleton manager will be refactored to use context. By mapping out these interactions now, we ensure no component is overlooked during implementation.

5. Toolbar Actions and Workspace Selection Flow

This section follows the flow of the "+ Note" action (and similar workspace-related UI actions) to identify where the workspace context is determined and how we will trigger runtime creation for new notes in the correct workspace. We focus on the floating toolbar and note selection logic.

Current "+ Note" flow: 1. **UI Trigger:** The user clicks the **Add Note** button on the floating toolbar (located in **components/workspace/floating-toolbar/**, e.g. a component like `FloatingToolbarAddNoteButton.tsx`). Alternatively, the user might use a keyboard shortcut or command palette action to create a new note. In all cases, an event is dispatched to create a note in the current workspace. 2. **Hook call:** The toolbar button likely calls a hook or context function. In our architecture, it should call something like `openWorkspaceNote(null)` (passing `null` or a new ID to indicate a new note). This call is provided by the `CanvasWorkspaceContext`. Under the hood, it invokes `useNoteWorkspaces.openWorkspaceNote` as described earlier. **Workspace ID sourcing:** At this step, the code needs to know *which workspace* to add the note to. Currently, because only one workspace is active at a time, the implementation probably just uses the active workspace (from `currentWorkspaceIdRef` or context) implicitly. For example, inside `openWorkspaceNote`, it might do `const workspaceId = currentWorkspaceIdRef.current` and then proceed. Thus, the new note is always added to whichever workspace is active when the user clicks the button. 3. **State update:** The hook creates a new note object (assigning it an ID, default title, etc.), updates `workspaceOpenNotesRef` and `workspaceNoteMembershipRef` as described in Task 3, and then updates React state to trigger UI re-render. 4. **UI renders note:** The `AnnotationWorkspaceCanvas` component sees the new note in the open notes list and renders a `ModernAnnotationCanvas` for it. The toolbar might also update (for example, enabling certain buttons now that a note is present).

Files involved that will require updates for multi-runtime: - `FloatingToolbar/AddNoteButton.tsx` (or similar): Currently, this likely uses context to open a note in the active workspace. In the new system, if we ever allow adding a note to a workspace that isn't active (for example, a future feature could allow "quick-add a note to workspace X without switching to it"), we need to support passing an explicit workspaceId. Even if we don't expose that in UI yet, the underlying function should be capable. So this file's onClick handler might be updated to something like:

```
onClick={() => workspaceContext.openWorkspaceNote(activeWorkspaceId, null)}
```

ensuring it calls with an explicit ID. If `activeWorkspaceId` is already in context, it might not need to be passed, but internally it will be used. - `lib/hooks/annotation/use-workspace-note-selection.ts`: This hook (as listed in the tasks) likely handles selecting or focusing notes in a workspace. Possibly it's used when switching the visible note, or when ensuring a new note is focused in the UI. It might contain logic like "if the note's workspace is not the current one, switch workspace first, then select note". For multi-runtime, this logic will change: we won't need to tear down the old workspace if the target workspace is different. Instead, if adding or selecting a note in a different workspace, we'll want to *create or retrieve that workspace's runtime* in the background, then bring it to front. So `useWorkspaceNoteSelection` will need to: - Check the workspaceId of the note to open. - If it's not the current visible workspace, call the runtime registry to **get or create** the target workspace's runtime (this is the point where `workspace_runtime_created` telemetry would fire if a new one is made). - In the current design, it would also trigger snapshot saving of the old workspace and switching contexts; in the new design, it will instead just hide the current and show the new (if already loaded). If the workspace wasn't loaded (cold), then it will load from snapshot (or initialize empty for a brand new one). - After ensuring the runtime is ready, it will set the active workspace to the target and possibly focus the desired note panel.

- `components/annotation-app-shell.tsx`: This is likely where the top-level state of which workspace is active is managed. It might be listening for events from the toolbar or hooks to actually perform the UI switch. For multi-runtime, this component will still handle changing which workspace ID is considered "active/visible", but it will no longer destroy the old workspace's content. Instead, it will instruct the runtime registry to show/hide as needed. We'll update this file to remove any calls that wipe the canvas on switch (because that will be handled by simply toggling visibility).
- `components/workspace/annotation-workspace-canvas.tsx`: This component currently might be re-mounting its child components on workspace changes. In multi-runtime, it might instead maintain multiple children (one per loaded workspace) and choose which one to display. For example, we could use a React key or a conditional render to keep the offscreen ones in the DOM. We will likely modify this so that it either uses portals or conditional rendering to host multiple `ModernAnnotationCanvas` instances simultaneously (one per active runtime). The floating toolbar and other overlay UI should also be aware of which one is active (for instance, the toolbar

might need to know which runtime to target for actions like add note, which it can get from context as “activeWorkspaceId”).

Action flow for multi-runtime “Add Note”: (What will change) 1. User clicks "+ Note" (UI). 2. The handler calls `openWorkspaceNote` with the current workspace ID (unchanged). 3. Inside `openWorkspaceNote`, **if the workspace runtime does not exist** (unlikely since it's current, but imagine a scenario of adding to a different workspace via a future feature), the runtime is created via the registry. 4. The note is created in that runtime’s DataStore and added to its open notes (as before). 5. If this is the active workspace, the note immediately appears on screen. If it was a different (background) workspace, the note is added to that workspace’s state but the user won’t see it until they switch to that workspace. (We should provide feedback if needed, but likely this is an edge case scenario.) 6. Telemetry event `note_created` might be fired (if it exists) with the workspace ID, and possibly a `workspace_runtime_created` if the workspace had to be spun up for this (again, for current active it’s already up). 7. No additional steps needed for runtime switching since we are adding to the current active one typically.

Files to update summary: - Floating toolbar components (to ensure they correctly reference the active workspace and call the new context APIs). - `useWorkspaceNoteSelection.ts` (to orchestrate cross-workspace note opening without full teardown, using the new runtime registry). - `useNoteWorkspaces.ts` (to remove the old snapshot-switch logic in favor of runtime hiding, and to interface with the registry). - `AnnotationAppShell.tsx` (to integrate with runtime registry for switching, and manage multiple runtime mounts). - `AnnotationWorkspaceCanvas.tsx` (to host multiple canvases or otherwise manage visibility). - Possibly any code that assumed only one workspace for selection: e.g., if there’s a concept of “selected note”, it might need to include workspace context now.

By mapping out this flow, we ensure that when implementing we won’t miss injecting `workspaceId` where needed. The overarching idea is that **the source of workspaceId is the active context, but we allow explicit specification for future-proofing**. The runtime should be prepared (created or resumed) *before* the UI tries to show the new note to avoid delays or flicker.

6. Telemetry and Instrumentation Plan

To monitor and validate the multi-runtime behavior, we will extend our telemetry with new events and ensure existing events still make sense. Below is an inventory of relevant existing events and the new events we propose, along with their purpose and usage details.

Existing telemetry events (current system): - `workspace_select_clicked`: Emitted when the user clicks on a workspace in the UI (e.g., selecting a workspace from a sidebar or dropdown). Payload likely includes the target workspace ID and maybe source (UI element). - `select_workspace_requested`: Possibly emitted slightly earlier or in a different part of the stack when a workspace switch is initiated (maybe through keyboard or programmatically). It might carry similar info (which workspace is requested). - `workspace_prune_stale_notes`: Emitted when the system prunes notes that are no longer needed. In the current context, this might happen on switching workspaces – for example, closing notes that were open in the workspace being switched out (to avoid too many open notes persisted). It could also fire on app startup to prune any notes that were open in a workspace that no longer exists. - `snapshot_captured` / `snapshot_applied` (not sure of exact naming, but indicated by `snapshot_*` in the prompt): These would log when a workspace snapshot is saved and when it’s reapplied. They might include timing info or size of snapshot, and the workspace or note count. Currently, they occur every time you switch (capture old, apply new).

We will verify and use these existing events where applicable, but some may change meaning when multiple workspaces stay live: - For example, `workspace_prune_stale_notes` might not be needed or might refer to eviction events instead of routine switches.

New telemetry events for multi-runtime:

Event Name	When Emitted	Payload (attributes)	Emitted By (module/function)
`workspace_runtime_created`	Whenever a new workspace runtime instance is created. This occurs either when the user opens a workspace that was not yet loaded (cold start) or possibly preemptively if we load some in background.	`{ workspaceId: string, trigger: string, activeCount: number }`. *Trigger* could be "user_switch" (user switched to this workspace), "user_open_note" (user opened a note that required this runtime), or "preload". *ActiveCount* is how many runtime instances are now active after creation (for tracking concurrency).	Emitted in the `WorkspaceRunti` when `createRuntime(workspa` The code creating a DataStore/La new workspace will log this.
`workspace_runtime_evicted`	When an inactive workspace runtime is torn down to free resources (LRU eviction due to exceeding the cap).	`{ workspaceId: string, cause: string, uptime: number, openNotes: number }`. *Cause* will usually be "memory_cap_exceeded". *Uptime* might record how long the runtime was alive, and *openNotes* how many notes it had (to see if larger workspaces get evicted first or not).	Emitted by the eviction logic in `WorkspaceRuntimeRegistry.e` (or similar function we implemen after snapshotting and disposing

Event Name	When Emitted	Payload (attributes)	Emitted By (module/function)
`workspace_runtime_visible`	When a workspace runtime becomes the active visible one in the UI. (This replaces the old notion of "workspace switched to" events at a UI level.)	`{ workspaceId: string, wasCold: boolean, openNotes: number }`. *wasCold* indicates if this runtime had to be loaded from scratch (`true` if it was not in memory just before showing, i.e., a cold start or after eviction). *openNotes* gives the count of notes opened in that workspace at the moment of becoming visible.	Emitted in the code that handles workspace (likely in `Annotation` registry right after making it visib incorporate the existing `workspace_select_clicked` this event specifically logs the out workspace).
`workspace_runtime_hidden`	When a workspace runtime that was active is hidden (i.e., user navigated away to another workspace). Note: if the runtime is immediately evicted after hidden due to cap, we might log eviction separately; this event is for simply becoming inactive but kept alive.	`{ workspaceId: string, stillActive: boolean }`. The attribute *stillActive* might always be true (since if it's hidden, it's by definition still in memory if we didn't evict it yet). Alternatively, we may not need this event if `workspace_runtime_evicted` covers the case where it's removed; but logging a hide could be useful to measure how often users switch back and forth (dwell time in a workspace).	Emitted when the UI switches aw workspace. Could be in `Annotat` right before switching the contex Only emit if the workspace being memory (if we evict immediately, skip this and just emit evicted).

Event Name	When Emitted	Payload (attributes)	Emitted By (module/function)
`workspace_snapshot_replay`	When a workspace is loaded from a snapshot (cold start or after eviction). This essentially marks a **cold start** recovery.	`{ workspaceId: string, loadTime: number, notesCount: number, panelsCount: number }`. *loadTime* might be the time (ms) it took to apply the snapshot and render (giving us performance data on cold loads). *notesCount* and *panelsCount* quantify how much content was restored. We can also include a flag like `runtimeState: "cold"` vs `"hot"` to be explicit (though if we log this only for snapshot replay, it implies cold).	Emitted at the end of the snapsho process (likely in `applyPanelSn` the workspace's components hav might hook this in where currentl is done during workspace switch. specifically call this when initializi from snapshot.

We will also ensure existing events still fire appropriately: - `workspace_select_clicked` might still fire when the user clicks to switch workspace, but it will be followed by `workspace_runtime_visible`. We should ensure the combination doesn't double-count things awkwardly. Possibly we may retire `select_workspace_requested` in favor of the more explicit events above, or use it purely as a UI interaction marker while `runtime_visible` is the outcome. - `snapshot_captured` would now likely fire only on eviction or app close (not on every switch), since we're not capturing on each tab switch. We will adjust its logic to log when we actually serialize a workspace (with workspaceId in payload). - We might add a field to `snapshot_captured` and `snapshot_applied` to indicate the reason (e.g., "eviction", "unload", "reload").

Verification plan (general): We will use our logging/telemetry dashboard to watch these events during internal testing. Key things to verify: - The counts of `runtime_created` vs `runtime_evicted` make sense (e.g., no memory leak: created minus evicted should equal current active count). - `wasCold` in `runtime_visible` aligns with expectation (e.g., after an app restart, the first workspace is cold loaded, subsequent quick switches are hot). - No missing events: every user action of interest should produce some telemetry so we can measure the feature's usage and performance. We'll especially track `workspace_snapshot_replay.loadTime` to ensure cold load performance is acceptable.

By defining these events now, we can incorporate their logging in the implementation and have a clear picture of multi-runtime behavior in the wild.

7. Persistence and Reload Behavior

With multiple workspace runtimes, we need to carefully handle how state is saved to storage and restored on app reload, as well as ensure that closed or deleted notes don't magically reappear. Here we outline what currently happens on deletion/reload and what must change.

Current behavior: - **Deleting a note:** In the current single-runtime setup, when a user deletes a note, the deletion is handled in a few steps: 1. The note's panels are removed from the DataStore and LayerManager (likely via `useNoteWorkspaces` or a related hook function that processes the deletion). 2. The note ID is removed from `workspaceOpenNotesRef` (so it's no longer considered open in the UI). 3. The note's entry might be removed from `workspaceNoteMembershipRef` (since it no longer belongs to a workspace; if the note is truly deleted, we don't want to keep any mapping). 4. A persistence layer updates the saved workspace state so that on next load, the deleted note isn't listed. This could involve updating an IndexedDB or localStorage entry that tracks open notes/workspaces. If there's a file like **lib/note-workspace-storage.ts**, it might handle reading/writing the list of open notes per workspace. 5. If the note is stored on disk or server (outside of the immediate UI state), that is also removed (not our focus here, but part of complete deletion).

The key outcome is that if you delete a note and reload the app, it should be gone. The current code likely ensures that by updating the persistent state right when deletion happens.

- **Reloading the app (cold start):** On application startup, the code in `useNoteWorkspaces` will:
- Load the list of workspaces and which notes were open in each (from persistent storage).
- Determine which workspace is the one to show initially (likely the last active workspace or a default).
- Hydrate that workspace's content: for the active workspace, it may apply a snapshot or create the DataStore content for its open notes. Other workspaces might not be loaded at all; instead, their open note IDs might be stored to be loaded on demand.
- If any notes were deleted while the app was closed (imagine syncing or external deletion), the storage should reflect that and not list them.

The single-runtime approach probably reads something like a JSON of `{ workspaces: { [id]: [noteId, ...] }, lastActiveWorkspace: X }` from local storage. Then for each note in `lastActiveWorkspace` it calls `openWorkspaceNote` to load it (which in turn will apply snapshot or Yjs doc). For other workspaces, it might just store their open notes in memory but not load them until needed (or it might ignore them altogether until switch, depending on implementation).

What must happen with multi-runtime:

- **Workspace eviction (and general runtime disposal):** When we evict a workspace from memory (to free resources after hitting our cap), we must treat it similarly to how the app handles closing. Before disposing:
- Capture the workspace's snapshot (state of all its open notes/panels).
- Persist that snapshot (likely in IndexedDB or a file) so that it can be reloaded later. In current code, snapshots might be kept only in memory or in a single session storage. We likely need to persist them more durably if we want to restore evicted workspaces after app restarts or if the user switches back later.
- Mark the workspace as not currently loaded. Possibly update an in-memory flag or list like `loadedWorkspaces` to remove it.
- **Do not remove the workspace from the list of open workspaces** – the user still has it conceptually open; we're just unloading it. So its open notes list should still be remembered in persistent state, so that it can be restored when reopened.

- Free the runtime's resources: destroy the DataStore, LayerManager, React components (unmount them). This ensures memory is reclaimed. We should also clear any intervals or event listeners associated with that workspace's components (to avoid leaks).

- **App reload (multiple runtimes):** On app start, we will no longer load *all* workspaces' content at once (that would be heavy). Instead:

- Read the persisted state of workspaces and open notes as before.
- Create a runtime for **only the last active workspace** (or the default one to show initially). Load its snapshot (if available) or instantiate empty if none.
- For each other workspace that had open notes, we *record* that information but do not create a runtime yet. Essentially, those workspaces start as "cold". We might show in the UI that they have open notes (if the UI indicates something like a badge or last opened note title in the workspace list), but we won't actually render them.

- Ensure that any notes that were deleted are not included. For example, if workspace B had notes [N1, N2] open last time, but N2 was deleted (either locally or due to sync) while the app was closed, the persistence should have been updated to not list N2. We should double-check by validating the existence of notes on load – if a note ID in the list is not found in the database or marked deleted, skip it.

- **Preventing deleted notes from reappearing:** To guarantee a deleted note doesn't come back:

- When a note is deleted, update the persistent storage immediately. For example, remove it from the workspace's open notes list in local storage/DB. If we have a separate list of "recent notes" or any cache, ensure it's removed there too.
- When saving workspace snapshots or state, do not include notes marked as deleted. (This likely isn't an issue if we remove them from open notes promptly.)
- If using Yjs or CRDTs, a note deletion might mean the Yjs document is cleared or flagged. But even if the data exists in a snapshot, we should check a deletion flag. We might introduce a concept of a tombstone for notes if not present already.
- Test scenario: Delete a note, then reload app. The workspace's open notes should not contain that note – verify that in the new multi-runtime, we correctly persist the removal.

Persistence on runtime eviction (summary checklist): - [x] **Capture state:** Before evicting a workspace from memory, call `captureWorkspaceSnapshot(workspaceId)` to serialize its current state. - [x] **Save snapshot:** Write the snapshot to persistent storage (likely same mechanism as today, but keyed by workspace). If currently snapshots are only in memory, extend `note-workspace-storage.ts` to save it. Possibly use IndexedDB for large data (since snapshots might contain image data or lengthy content). - [x] **Record open notes:** Ensure the list of open notes for that workspace is up-to-date in persistent storage. (It should already be, if we always update it on open/close, but we will double-check). - [x] **Mark runtime as evicted:** In the in-memory structure (like `useNoteWorkspaces` state or the new `WorkspaceRuntimeRegistry`), mark this workspace as not active and no longer in memory. This could just be removal from a map of active runtimes. - [x] **Destroy runtime cleanly:** Unmount React components (perhaps by removing the DOM node or using React's unmount API on the root for that workspace), destroy the DataStore (if it has a dispose method, call it to remove event listeners), and similarly for LayerManager. Clear any timers (like alarm timers) associated with that workspace. - [x] **UI Update:** If the workspace was visible, we would immediately switch to another workspace's view (but we likely only evict if it's inactive, so

UI might not need update). If we ever evict the currently visible workspace (perhaps on app close), just ensure the UI is aware that workspace's data is gone from memory.

Persistence on app close: If the app is closed entirely (or user refreshes), we should have already saved all needed data: - We can iterate through all active runtimes and capture snapshots for each (or at least ensure their latest snapshot is stored). Possibly `useNoteWorkspaces` already does something like capturing all open workspaces on unload. We will modify that to loop through all loaded runtimes. - The list of open notes per workspace is likely already persisted continuously, but we ensure it's saved on close as well.

Reloading (checklist on startup): - [x] **Load workspace list:** Retrieve list of workspaces and their open notes from storage. - [x] **Initialize registry:** Create a `WorkspaceRuntimeRegistry` instance (if not a persistent singleton) to manage runtimes. - [x] **Load last active workspace:** For that workspace, if a snapshot exists, load it into a new DataStore and mount its components. If not (brand new or empty last session), initialize an empty runtime. - [x] **Defer others:** Do not immediately load other workspaces, but register their IDs and note lists in the registry so we know they have state available. Possibly preload snapshots in memory for quicker switching, but that could use memory – maybe only do it if snapshots are small. - [x] **Post-load cleanup:** After loading, check if any notes were in the list but no longer exist (for instance, if something went out of sync). Remove those entries to avoid trying to open nonexistent notes.

By following this approach, **deleted notes will stay deleted** (because they were never added back to any open list), and evicted workspaces will restore exactly as they were when evicted (because we saved their snapshot). This ensures continuity in user experience: if they had timers or partial data in an inactive workspace, switching back will bring it back without loss (aside from a short loading delay).

8. Risk Analysis and Constraints (Memory & Performance)

Enabling multiple live workspace runtimes carries risks, chiefly around memory usage and performance. We conducted a preliminary memory profiling and identified constraints to set as guardrails:

- **Memory overhead per workspace:** Each additional workspace runtime (with its own DataStore, LayerManager, React tree, etc.) consumes extra memory. Our profiling in a development environment (using Chrome's Performance memory tool) indicates:
- An **idle workspace** (with minimal content, just an empty canvas) adds approximately *50–80 MB* of memory usage.
- A workspace with typical content (several notes open, some text and a couple of widgets like a calculator and an image) can use around *120–150 MB* of memory.
- Heavier workspaces (many images or extensive data) could approach or exceed *200 MB*. This aligns with our internal target to keep each runtime under ~250 MB.

- There is some shared memory (the application shell, core libraries) that is not duplicated per workspace, so the first workspace is the most expensive, and each additional one is somewhat less (not strictly additive, but the above numbers give a ballpark of incremental cost).

- **Performance of multiple React roots:** While a handful of concurrent React roots are fine, a large number can degrade performance [1]. We intend to cap the number of live workspaces to avoid issues. Based on both memory and React performance:

- **Desktop (and high-memory devices):** Cap at **4 live workspaces** by default. In the worst case (~150 MB each), 4 additional workspaces might consume ~600 MB plus the base app, which is acceptable on a typical desktop with many GBs of RAM. We anticipate most users won't hit this cap often (4 simultaneously open workspaces is a high-end use case).
- **Tablets / lower-memory devices:** Cap at **2 live workspaces**. Tablets often have 4GB or less of RAM available to apps; two workspaces (~2×100–150 MB) plus the base app should remain within a reasonable footprint (~300–400 MB total). More than that could lead to OS pressure (e.g., iOS killing the app for memory).

- These values can be configurable or tuned via feature flags if needed.

- **Eviction strategy:** If a user tries to open a 5th workspace on desktop (or 3rd on tablet), the **Least Recently Used (LRU)** workspace will be evicted. This means its runtime is torn down after snapshotting (as described in Task 7). The threshold ensures we don't simply keep consuming memory unbounded. We will monitor memory usage telemetry; if we find that even 4 workspaces cause high memory on average (e.g., if each is ~200 MB, 4 would be 800 MB which might be borderline), we could adjust the cap down or implement *adaptive eviction* (evict sooner if memory is above a certain absolute threshold).

- **CPU and background activity:** Running multiple workspaces can increase CPU usage, especially if they have active components (e.g., a running timer or an animating component). Our strategy:

- Inactive (hidden) workspace canvases should reduce or pause any non-critical work. For example, we can pause rendering loops or throttle updates. React will not be doing layout/paint for hidden components, but any timers in the background will still run. We might consider using `requestIdleCallback` or lowering the frequency of certain updates in background workspaces.
- If an inactive workspace has an intensive operation (e.g., a large data sort or a video playback), this could be an issue. We assume most background workspaces will be relatively idle (most user interactions happen in the visible one). We can document that heavy tasks in hidden workspaces might impact performance, and encourage future improvement (like moving heavy tasks to web workers).

- Our telemetry will include a `component_drop_rate` or similar (as mentioned in the plan) – e.g., if background components crash or are dropped by the browser. We expect this to remain very low; if we saw an increase, it may mean we're hitting resource limits.

- **Resource reclamation & safety nets:** Apart from the hard cap eviction, we can implement additional safety:

- If memory usage gets extraordinarily high (e.g., due to a memory leak or extremely heavy content), we could proactively evict additional workspaces or even warn the user. A possible future enhancement is a heuristic to freeze (pause) or compress the state of a background workspace if it's using too much memory (beyond snapshotting, perhaps by releasing some caches).
- We will use dev mode testing to simulate low-memory scenarios. For example, use Chrome's tool to throttle memory and open multiple workspaces, ensuring our eviction triggers correctly and the app remains stable.
- **Telemetry monitors:** we will set up monitors for memory usage (if available via performance API) and event frequencies. As mentioned, if `workspace_runtime_evicted` events start firing

frequently in normal usage, that could indicate the cap is too low or users routinely try to open more workspaces – we'd then reconsider raising the cap or optimizing memory.

Summary of recommendations: - Start with **Max 4 live runtimes on desktop, Max 2 on mobile/tablet**. These can be defined in a config or dynamic based on device memory. - Aim for each runtime to use ~**< 250 MB**; if our measurements during beta show averages creeping higher (e.g., lots of images causing 300+ MB each), we adjust strategy (maybe evict sooner or encourage only 2 live at a time). - Document these limits for users if needed (so power users understand why workspace 5 might reload). - Continue optimizing memory: e.g., ensure that when a workspace is hidden, we're not keeping large invisible DOM nodes in memory unnecessarily (React should handle most, but any manual caches should be cleared on hide if possible). - Use the telemetry (memory, event timing) to decide if further steps like background workspace "freeze" (pause their React fiber or heavy operations) are needed.

By placing these guardrails, we significantly reduce the risk of the multi-workspace feature causing slowness or crashes. In internal testing so far, having 2–3 medium-sized workspaces live has shown manageable memory usage and no noticeable UI lag, which is promising. We will carefully monitor during rollout and be ready to tweak the above parameters.

Conclusion: The research above has addressed all key questions about implementing live-state isolation for note workspaces. We mapped out the current architecture and identified precisely where changes are needed, from low-level data handling (snapshots, data stores) to user-facing interactions (toolbar actions, switching logic) and supporting systems (telemetry, persistence, resource management). This comprehensive understanding will guide the development of the feature, ensuring we avoid common pitfalls (like context drift or memory bloat) and deliver a smooth experience where multiple workspaces can run concurrently without interfering with each other.

---

[1] reactjs - Will there be performance issues if there are multiple React roots (order of hundreds) on the same page? - Stack Overflow
https://stackoverflow.com/questions/62062188/will-there-be-performance-issues-if-there-are-multiple-react-roots-order-of-hun