ChatGPT

Hypothesis Validation

1. **Race Condition (Branch loader vs. hydration):** The code shows that opening a branch panel (`handleCreatePanel`) does not wait for the hydration to finish. There is no check on `hydrationStatus` before loading a branch. In fact, the branch loader attempts to retrieve panel data from the `branchesMap` immediately, and if not found it logs a warning and aborts [1]. This means if a user triggers a branch panel open while the layout hydration is still in progress (or before the branch data is seeded), the loader may run too early. In such a case, `branchesMap.get(panelId)` returns nothing (since hydration hasn't populated it yet), leading to "No data found for panel X" warnings [2]. This confirms a race condition: the branch panel might fail to open or use incomplete data if hydration hasn't finished. Additionally, if the branch panel *was* persisted (exists in the database) but hydration hasn't applied it to the stores yet, the loader could mistakenly treat it as a new panel and call a create API. There is no guard against double-loading or duplicate creation in that window, since the code only checks if the panel is already in the React state (`canvasItems`) – not if it exists in the backend [3]. In summary, the branch loader can indeed run out of order, causing potential duplication or missing data issues due to a race with hydration.

2. **Coordinate System Confusion (screen-space vs. world-space):** The implementation is intended to use **world-space** coordinates for all stored panel positions [4] [5]. However, the branch loader logic appears to mix coordinate spaces. When a new branch panel is opened with a `parentPosition`, the code directly assigns `panelData.position = parentPosition` without any conversion [6]. The comment even says "Update position to be beside parent," implying `parentPosition` was likely provided in *screen-space* (e.g. an on-screen offset next to the parent panel). Yet this value is being stored in `panelData.position`, which by contract should hold a world coordinate. Subsequently, the code computing the final panel position for rendering assumes that any `branchData.position` is a world coordinate – it calls `worldToScreen(branchData.position, camera, zoom)` if a position exists [7]. If in reality `branchData.position` was set using an on-screen pixel offset, this conversion is incorrect (converting screen→screen again). This confusion would cause the panel to appear at the wrong location. For example, treating a screen coordinate as world would add the camera translation a second time [8]. In effect, a branch panel might **jump** or be saved to an incorrect position because of this mix-up. The code supports this: the branch loader prioritizes `branchData.position` over `worldPosition` for placement [9], so if it stored a screen-space value in `position`, the usage of `worldToScreen` would miscompute the panel's screen location. This explains why branch panels were not reappearing where they were last placed – their coordinates were being misinterpreted due to inconsistent coordinate space usage.

3. **Branch Loader Overwrites Hydrated Data (worldPosition lost):** During hydration, each panel's data is seeded with both a `position` (world-space) and a duplicate `worldPosition` field to explicitly mark that coordinate [10]. This means after hydration, `branchData.worldPosition` should equal `branchData.position` (the true world coords). The branch loader, however, does **not** preserve that field when it runs. If a `parentPosition` is provided, it overrides `panelData.position` but never sets `worldPosition` [11]. In a persisted branch's case, this

effectively **discards** the stored world coordinate: the object still has the old `worldPosition` from hydration (or none at all if hydration hadn't run), now mismatched with a new `position`. The debug logs confirm this mismatch. The loader logs `hasWorldPosition` as false in many cases [12], meaning the branch data in memory ended up with no `worldPosition` field (or an outdated one) after the loader ran. Thus, the hydrated world-space data was overwritten or ignored. In practice, this would mean the branch panel's position change (from the loader) wasn't recorded in the world-space field used for persistence. If the user never moved the panel afterward, the new position might never get persisted properly. On reload, the panel could revert to the old coordinate (or default), indicating a **failure to persist the branch panel's position**. This hypothesis is strongly supported by the code: the design contract says stores must hold world coordinates [13], yet after the branch loader runs, the branch's data store entry violates this (no correct `worldPosition`). Essentially, the loader's update clobbers the synced world coordinate with a transient screen-space value, leading to lost or non-persisted position data.

4. **Different Persistence Code Paths (branch vs. main):** There are indeed subtle differences in how main panels and branch panels are persisted:

5. The **main panel** is always present by default and, on first load, if it's missing from the database, the app calls `persistPanelCreate` to save it [14] [15]. The main panel's default position is set in world coordinates (2000,1500) intended to appear centered given the initial camera offset. Notably, this default goes through the same `persistPanelCreate` flow which treats the provided position as screen-space and converts it to world before saving [16] [17]. Because the main panel and camera defaults were chosen consistently, this doesn't cause a visible issue for main. The main panel's data is also present in the `dataStore` from the start (ensuring it always has a `worldPosition` after hydration or creation).
6. For **branch panels**, the persistence flow is triggered differently. A new branch panel is created via the event/loader path, which calls `persistPanelCreate` immediately after adding the panel to the state [18]. However, unlike the main panel, the branch's state in the stores is not fully prepared when calling `persistPanelCreate`. As noted, the branch's `position` might be in the wrong space and `worldPosition` missing at that moment. Moreover, `persistPanelCreate` does not use the same transactional update to the in-memory stores that `persistPanelUpdate` does. The update path (`persistPanelUpdate`) explicitly queues updates to all stores (dataStore, branchesMap, layerManager) with both `position` and `worldPosition` fields kept in sync [19] [20]. The create path, by contrast, sends the data to the backend but does not update the local stores at all (it relies on the hydration or the existing `branchesMap` content) [21]. This means a branch panel might be created in the backend with a world-coordinate, but the frontend's store still lacks a `worldPosition` field until either the user moves the panel (triggering an update transaction) or a full re-hydration occurs. In short, the main panel's position persistence goes through a safer path (always ending up with world coords in store), whereas branch panels used a path that left a gap between creation and consistent storage. This difference in code paths contributed to branch panel positions not persisting reliably.

Root Cause Analysis

All four factors above intertwine, but the primary root cause is the **mismanagement of coordinate state for branch panels**, compounded by timing issues. The branch panel loader was not respecting the architectural contract that **"all client stores hold world-space position values"** [13]. By writing a likely

screen-space position into the store without updating the world-space field, it introduced a state inconsistency. This led to incorrect conversions (screen vs world) and ultimately meant that the panel's true world coordinate was never properly persisted to the backend.

In practical terms: when a branch panel was opened, the system might place it on screen next to its parent, but fail to record that placement in the canonical world-space form. If the panel was not moved (no drag to trigger an update), the backend either never got the position or got an incorrect one. Upon reloading the note, the hydration would either use the old position or none at all, causing the panel to reset location (the **persistence failure**). The race condition exacerbated this by sometimes running the loader before hydration data arrived, meaning branch panels could be placed using stale or default info and even trigger duplicate saves. However, the core issue to fix is ensuring that whenever a panel's position is set or used, the world-space coordinate is correctly maintained.

Specific Fix Proposal

To resolve the issue, we need to ensure branch panels always store and persist their positions in world-space, just as the architecture prescribes [4]. The fix should be minimal and apply only to branch panel handling. The key changes are:

- **In the branch loader (`handleCreatePanel` in** *annotation-canvas-modern.tsx*): **when a `parentPosition` is used to position a new branch panel, convert that position to world coordinates before updating the store, and set both `position` and** `worldPosition`. This keeps the in-memory data consistent and correct for persistence. We should also refrain from overwriting a panel's position if it already has a `worldPosition` (i.e. if it was hydrated/persisted before), so that a reopened panel appears where it was saved rather than being forced next to the parent again.

Below is a surgical code change illustrating the fix:

```
@@ function handleCreatePanel(panelId, parentPanelId, parentPosition) {
-    if (parentPosition && panelData) {
-        panelData.position = parentPosition;
-        branchesMap.set(panelId, panelData);
-    }
+    if (parentPosition && panelData) {
+        // Convert screen-space parentPosition to world-space (ensures
consistency)
+        const camera = { x: canvasContextState.canvasState.translateX || 0,
+                         y: canvasContextState.canvasState.translateY || 0 };
+        const zoom = canvasContextState.canvasState.zoom || 1;
+        const worldPos = screenToWorld(parentPosition, camera, zoom);
+        panelData.position = worldPos;
+        panelData.worldPosition = worldPos;
+        branchesMap.set(panelId, panelData);
+    }
```

File: `annotation-canvas-modern.tsx` – **Lines 150–158** (within `handleCreatePanel` for the non-plain branch case) are updated to compute `worldPos` from the given `parentPosition` using the current camera state, and to store this value as both the panel's `position` and `worldPosition`. A similar update should be made in the plain-mode branch of this function (using `dataStore` instead of `branchesMap` for storage) so that the logic is consistent [22] [23]. In plain mode, we can obtain the same `canvasContextState` for camera values, or assume default if unavailable.

- **Import the needed utility:** Ensure that the `screenToWorld` conversion function is imported from the coordinate utils. In the same file, add `screenToWorld` to the import statement at the top (where `worldToScreen` is already imported) [24] [25].

With this change, branch panels will honor the world-coordinate contract. The parent-provided position will be translated into the global canvas coordinate system before use. As a result, the internal stores will have `worldPosition` set for branches at creation time, just like for hydrated panels. This directly addresses Hypothesis 2 and 3: the coordinate is no longer confused (we store the correct world value), and the `worldPosition` field is no longer dropped on the floor.

Why this fix works: After the fix, whenever a branch panel is opened or positioned: - The **stores always get a world-space coordinate** (fixing the consistency issue). The panel's data in `branchesMap`/`dataStore` will look just like a hydrated or updated panel – e.g. `{ position: {x: 3200, y: 1800}, worldPosition: {x: 3200, y: 1800}, ... }`. The `hasWorldPosition` flag will be true, indicating the data is complete. - The next time the panel is rendered or moved, all calculations are correct. For instance, the `worldToScreen` call in the render logic will receive a proper world coordinate and convert it to the right screen position [26]. And if the user drags the panel, the persistence hook will update both fields atomically (which it already does) [19], so subsequent loads will get the right data. - This fix is scoped to branch panels. It doesn't affect main panel logic except to make it consistent. (Main panels were already handled in world-space after hydration; we are essentially extending that guarantee to branch panels.) - We uphold the architectural contract from *implementation.md*: world-space coordinates are the single source of truth for panel positions [4], and we perform the necessary conversion at the point of UI action. By doing so, all panels – main or branch – persist their positions in a zoom-independent, camera-independent way. On reload, the canvas will hydrate these world coordinates and render the panels exactly where they were, finally resolving the branch panel position persistence failure.

Sources:

- Implementation design for coordinate system and state persistence [4] [13]
- Hydration code seeding world coordinates in stores [10]
- Branch loader code before fix (showing race and missing worldPosition) [11] [9]
- Panel persistence hook ensuring world-space updates on drag [19]

---

[1] [2] [3] [6] [7] [9] [11] [12] [14] [15] [18] [22] [23] [26] annotation-canvas-modern.tsx
file://file-NWi5HC85eJjAu8gJpHLtYp

[4] [5] [13] implementation.md
file://file-GxcaWGi2Vi5rYGphgpssyw

8 24 25 coordinate-utils.ts
file://file-HwKr8DJYnnKHEN98EpykYH

10 use-canvas-hydration.ts
file://file-3b2qzBMh8aC3KGh2hRF12J

16 17 19 20 21 use-panel-persistence.ts
file://file-9NerPsGnKxjTxz291vesZ2