ChatGPT

Phase 6 Deprecation Removal Plan – Codebase Validation

Phase 6.1: CurveDataStore Removal (CRITICAL)

Existence of Legacy Store: The codebase still includes the `CurveDataStore` class in `stores/curve_data_store.py`, explicitly marked as a **deprecated compatibility layer** [1] [2]. This class defines multiple Qt signals (`data_changed`, `point_added`, `point_updated`, etc.) and manages a single curve's data in-memory [3] [4]. We confirm that `CurveDataStore` is indeed present and used: for example, `CurveViewWidget` holds a `_curve_store` instance via the `StoreManager` and uses it to serve its `curve_data` property [5] and `selected_indices` property [6]. Thus, all files and members mentioned for removal (the `curve_data_store.py` file and the `_curve_store` references in the UI) exist in the Phase 5 codebase.

Current Behavior vs Plan: In Phase 5, **ApplicationState** was introduced as the new single source of truth for curves, but **StateSyncController** currently bridges ApplicationState and the legacy store. The code shows that `StateSyncController.connect_all_signals()` hooks up **CurveDataStore's signals** to update the widget (e.g. `_curve_store.data_changed` -> `CurveViewWidget._on_store_data_changed`) [7] and also connects **ApplicationState signals** to handlers that sync changes back to the store [8] [9]. Notably, `_on_app_state_curves_changed` in StateSyncController explicitly copies ApplicationState's `__default__` curve data into the `_curve_store` to keep the old `widget.curve_data` in sync [9] [10]. The plan's list of "10 methods removed" corresponds exactly to what we see: all `_connect_store_signals` and `_on_store_*` handlers, the `_sync_data_service` helper, and the store-sync portions of the `_on_app_state_*changed` callbacks are present and would be deleted [11] [9]. We find each of these methods in the current code, confirming the plan's scope is accurate. For example, `_on_store_point_added/removed/updated` etc. all exist in **StateSyncController** and propagate changes to the widget and its signals [12] [13]. Removing them in Phase 6.1 is safe because their only purpose is to mirror data between ApplicationState and the soon-to-be-removed store.

References & Migration Targets: The plan notes ~41 files to update, which aligns with our observations. Many components still reference the store or its aliases. For instance, **CurveDataFacade** currently writes to both the store and ApplicationState in methods like `set_curve_data()` (updating `_curve_store` then calling `app_state.set_curve_data("__default__", data)`) [14] [15]. Controllers like **PointEditorController** and **ActionHandlerController** similarly use `_curve_store`: e.g. `ActionHandlerController._get_current_curve_data()` returns `self._curve_store.get_data()` as the "single source of truth" for saving files [16]. After Phase 6.1, these must be re-pointed to ApplicationState (e.g. use `app_state.get_curve_data(active_curve)` instead). The codebase state confirms the plan's migration targets are correct – for example, `PointEditorController` today listens for both state manager *and* store selection signals (it has `on_selection_changed` from StateManager and `on_store_selection_changed` for the legacy store)

[17] [18]. This dual-listening will be simplified once the store is gone. We also see that **StoreManager** currently connects CurveDataStore → FrameStore: whenever curve data changes, FrameStore recomputes the frame range from the store's data [19]. This dependency will need to shift to ApplicationState (or StateManager) after removal, which the plan acknowledges ("remove CurveDataStore management" in StoreManager and FrameStore). In summary, every file and reference cited in Phase 6.1 (CurveViewWidget, StateSyncController, CurveDataFacade, TimelineTabs, MainWindow, SignalConnectionManager, PointEditorController, MultiPointTrackingController, ActionHandlerController, StoreManager, FrameStore, stores `__init__.py`) exists and behaves as described. The Phase 5 code still treats `CurveDataStore` as the authoritative source for a single curve – the **post-Phase 6 architecture** of using only ApplicationState is not yet fully realized, which is exactly why this refactor is needed. We concur that **ApplicationState** is intended to be the "only source of truth," and the code supports this intent (ApplicationState holds all curves and emits `curves_changed` [20]), but currently the presence of the store means there is duplicate state. Removing the store and its sync overhead will bring the implementation in line with the plan's target architecture. No missing methods or files were found – all elements slated for removal are present. One minor point: the plan mentions removing store sync in `_on_app_state_active_curve_changed()`, but in the current code this handler already **only** updates the view (it doesn't sync to the store) [21], so there is effectively nothing extra to remove there beyond the method itself. This doesn't affect the plan's validity; it simply means that particular method is already largely inert regarding the store. Overall, Phase 6.1's removal plan is **consistent and safe** relative to the code: it will delete deprecated code paths that we verified are still in use for legacy support, and all new code has equivalent ApplicationState logic in place to take over.

Phase 6.2: Removal of `main_window.curve_view` Alias (HIGH)

Alias Definition and Usage: The code defines `MainWindow.curve_view` as a deprecated alias to the actual `CurveViewWidget`. In `ui/main_window.py`, we see `self.curve_view: CurveViewWidget | None = None  # Deprecated - use curve_widget instead` [22]. There is also a `set_curve_view()` method that simply assigns this attribute and logs a message for legacy reference [23]. The new code uses `self.curve_widget` as the primary attribute for the curve editor widget (initialized via the UI controllers), but during Phase 3–5 the old `curve_view` was kept for backward compatibility. The plan lists four production references to `curve_view`, and we have confirmed them: in `ui/menu_bar.py`, several menu actions check `if self.main_window.curve_view is not None` and then operate on it (e.g. **Select All**, **Deselect All**, **Delete Selected**, **Reset View**) [24]. In each case, the code obtains `curve_view = self.main_window.curve_view` and then calls the appropriate method in the **InteractionService** (which expects a `CurveViewProtocol`) [24]. Likewise, the **InteractionService** itself contains fallbacks for `main_window.curve_view`. For example, when updating points or resetting the view, it does things like: `if main_window.curve_view is not None: ... fallback to curve_view ...` in a few places [25]. We found the block around interaction_service line 1316 where it mirrors data to `curve_view.curve_data` (likely for legacy UI updates) [25], and checks around line 1350+ for calling `curve_view.set_points` or `curve_view.update()` if that older interface is present [26] [27]. These occurrences exactly match the plan's note of "6 fallback references" in InteractionService.

Consistency with Plan Intent: The plan's migration strategy is to remove the `curve_view` attribute entirely and use only `curve_widget`. This is well-supported by the current design: **MainWindow** already has `self.curve_widget` (set in UI initialization) and all new code references `curve_widget`. The alias

`curve_view` is effectively a duplicate pointer used only by older code paths. We verified that **MainWindowProtocol** (the interface used for type-checking) still includes `curve_view` for structural compatibility, but removing it will just require updating those protocols and any tests that set `curve_view`. Importantly, **every usage of `curve_view` in the code has an equivalent `curve_widget` ready**. For example, the MenuBar actions that currently check `curve_view` can check `curve_widget` instead, since in practice `curve_widget` is always non-None once the UI is initialized. Indeed, in the main window's constructor, `curve_widget` is declared and later populated (we see `curve_widget: CurveViewWidget | None = None` and it's assigned by the UIInitializationController) [28] [29]. The alias is even documented as deprecated in code comments. We do not find any hidden or additional uses of `curve_view` beyond those listed; it is not used for any logic except those UI event handlers. Tests also honor both names – many test fixtures create a mock MainWindow and set both `curve_widget` and `curve_view` to the same dummy widget for safety [30], which aligns with the plan's note that "all test mocks set both curve_widget and curve_view." This indicates that tests will need slight updates but are aware of the dual names.

Deprecation Status: The code does not emit a deprecation warning when `curve_view` is accessed (it's just a variable), but it is clearly labeled and logged as legacy. For instance, calling `main_window.set_curve_view(x)` logs "Legacy curve view reference set" [23]. This confirms the development team treated it as a temporary alias. Removing the `MainWindow.curve_view` attribute and its setter has no effect on core functionality – the **MainWindow** already uses `self.curve_widget` for all internal purposes (e.g. adding it to layouts, updating it, etc.). We also note that **InteractionService** and **MenuBar** already prefer the new widget when available: the InteractionService logic typically checks `main_window.curve_widget` first (since Phase 5), and only uses `curve_view` in an `elif` fallback for older scenarios [25]. The plan's Step 2 and Step 3 (updating MenuBar and InteractionService to drop those fallbacks) are perfectly in line with this.

In summary, each referenced file and line in sub-phase 6.2 exists and matches the plan. Converting those few `curve_view` references to `curve_widget` (and removing the now-unused `set_curve_view` function and protocol field) is straightforward. No discrepancies were found: the **MainWindow.curve_view** alias behaves exactly as described, and the new `curve_widget` is already the single authoritative reference to the curve editor widget in Phase 5. This sub-phase is low risk – essentially removing dead weight.

Phase 6.3: Removal of `timeline_tabs.frame_changed` Signal (MEDIUM)

Current Implementation: The **TimelineTabWidget** (timeline tabs UI) defines a signal `frame_changed: Signal(int)` which is marked as deprecated in the comment and docstring. In the code, at `ui/timeline_tabs.py` we see: `frame_changed = Signal(int)` with a note "DEPRECATED: Use StateManager.frame_changed instead for new code – kept for backward compatibility only" [31]. The TimelineTabWidget no longer uses this signal for its own operation; instead, it uses the StateManager's frame logic. Specifically, when a user clicks or scrubs the timeline, the widget calls `self.state_manager.current_frame = X`, and then emits its internal `frame_changed` *only* for legacy listeners, wrapping it with a DeprecationWarning. The code for setting the current frame illustrates this: in `TimelineTabWidget.set_current_frame()`, after delegating to StateManager, it does:

```
warnings.warn("timeline_tabs.frame_changed signal is deprecated...",
DeprecationWarning)
self.frame_changed.emit(frame)
```

[32]. We confirm this behavior: on any programmatic call to `set_current_frame`, `frame_changed` will emit with a warning once [33] [34]. However, in normal operation, the application no longer relies on this signal. The **SignalConnectionManager** explicitly does *not* connect `timeline_tabs.frame_changed` to anything, noting in code that doing so was removed to prevent circular updates [35]. Instead, **StateManager.frame_changed** is the single source of truth for frame changes.

Match with Plan: The plan states this signal exists with a DeprecationWarning and that ~10 references across 5 files need to be updated. Our findings align: aside from its definition and emission in TimelineTabs, the remaining references are likely in tests and possibly any leftover slots that listened to it. We searched for usage and found that production code has largely migrated away: for example, **MainWindow** does not connect to `timeline_tabs.frame_changed` anymore (older code might have, but it's gone now). Instead, MainWindow connects to **StateManager.frame_changed** (see `state_manager.frame_changed.connect(self.on_state_frame_changed)` in SignalConnectionManager) [36]. The **StateManager.frame_changed** signal in turn is emitted whenever ApplicationState's current frame changes (StateManager connects ApplicationState.frame_changed -> StateManager.frame_changed) [37] [38]. In effect, TimelineTabWidget now notifies the StateManager (which emits its frame_changed), and GUI components listen to StateManager. The deprecated `timeline_tabs.frame_changed` is effectively unused in the app's logic except possibly to support test code or external plugins.

We looked at tests and found assertions consistent with this plan. For instance, `tests/test_navigation_integration.py` sets up a QSignalSpy for `window.state_manager.frame_changed` and expects it to fire on user input, and then asserts that `timeline_tabs.frame_changed` **does not** fire unless explicitly invoked via `set_current_frame` (to avoid double-emission) [39] [40]. This confirms the intent: in "new architecture" usage, only StateManager's signal is used, and TimelineTabs' own signal is silent. In fact, the test explicitly checks that in normal navigation, `state_spy.count() > 0` (StateManager fired) but `timeline_spy.count() == 0` (TimelineTabs did not emit) [41] – and then they manually call `timeline_tabs.set_current_frame(5)` and expect one emission of `frame_changed` with a warning for that direct call [40]. This aligns with the plan's note that external code should switch to `state_manager.frame_changed`.

Conclusion for Phase 6.3: All evidence shows that `timeline_tabs.frame_changed` is still present as a deprecated alias, exactly as described. Removing it will involve: (a) deleting its Signal definition and any `.emit()` calls (there are only a couple), and (b) updating any tests or external usage. We did not find any lingering dependencies on it in the application logic – the StateManager pipeline fully covers frame change propagation. The plan's assumption that **StateManager.frame_changed is the correct API** is validated by the code: StateManager's `frame_changed` is emitted on frame updates and is connected to UI updates like MainWindow's on_frame_changed handler and background image updater [36]. Meanwhile, TimelineTabs emits `frame_changed` only with a deprecation warning, so nothing in production relies on it (the SignalConnectionManager even comments that the old connection was removed) [35].

There is no architectural inconsistency here: the **ApplicationState/StateManager** is indeed the single source of truth for current frame after Phase 5, and timeline's own signal is purely vestigial. We anticipate no issues removing it. The "10 references in 5 files" likely count test occurrences and the few spots in the timeline code itself; we haven't found any extra references beyond those. One possible minor catch: after removal, any test that was spying on `timeline_tabs.frame_changed` (like the one above) should be adjusted to spy on StateManager instead, but the plan already accounts for updating tests. In short, Phase 6.3 is consistent with the code and should be a clean removal of a now-unused legacy signal.

Phase 6.4: Removal of `CurveViewWidget.should_render_curve()` (LOW)

Current Behavior: The `CurveViewWidget.should_render_curve(curve_name: str) -> bool` method exists and encapsulates the **curve visibility logic** (based on each curve's metadata `visible` flag and the global display mode). We located this method in `ui/curve_view_widget.py` and verified it's annotated as deprecated. The docstring explicitly states it's deprecated in favor of using a pre-computed RenderState, and it details the old vs new patterns [42] [43]. The logic implemented in `should_render_curve` matches exactly what the plan describes: it first checks the curve's metadata visibility, then applies a three-branch filter for the display mode (ALL_VISIBLE, SELECTED, or ACTIVE_ONLY) [44] [45]. If `display_mode == ALL_VISIBLE`, it returns True for any curve that is visible; if SELECTED, True only if the curve is in the `selected_curve_names` set; if ACTIVE_ONLY, True only for the active curve [46]. This is the logic that Phase 6.4 intends to remove from per-curve checks in favor of computing it once.

Migration to RenderState: The codebase has already introduced the `RenderState` class (in `rendering/render_state.py`) and a corresponding method `CurveViewWidget.compute_render_state()`. We found that the rendering pipeline is already using this new approach. In `CurveViewWidget.paintEvent` (indirectly through `CurveViewWidget._optimized_renderer`), the code does:

```
render_state = self.compute_render_state()
self._optimized_renderer.render(painter, event, render_state)
```

[47]. The `compute_render_state()` method creates an immutable `RenderState` object representing the current frame, selection, and critically, it computes `visible_curves` as a frozenset of curves that pass the visibility filters [48] [49]. We inspected `RenderState.compute(cls, widget)` and confirmed it implements the same logic as `should_render_curve` but in a vectorized way: it iterates over all curve names in `widget.curves_data`, checks each curve's metadata visibility and the widget's display mode once, and builds the `visible_curves` set [48] [49]. This precomputation happens once per repaint instead of multiple times per curve. The plan's suggested replacement snippet (`render_state = RenderState.compute(widget)` and loop over `render_state.visible_curves`) is exactly what the code is doing now – except the code even avoids looping in Python by letting the C++ paint code iterate, using the `visible_curves` set as a filter.

Crucially, we searched for any active uses of `widget.should_render_curve` and **found none in the current rendering logic**. The `OptimizedCurveRenderer` no longer calls it; instead, it expects a

`RenderState`. In fact, `OptimizedCurveRenderer.render()` now takes a RenderState and doesn't need to query the widget about each curve. The documentation and comments in the code further confirm this shift. The plan's mention that `OptimizedCurveRenderer._render_multi_curve` used to call `should_render_curve` is likely referring to older code. In the current code, that renderer is decoupled: `CurveViewWidget` passes in the RenderState which already has the `visible_curves`. For completeness, we looked at the `should_render_curve` references: aside from its own definition and a mention in CurveDataFacade's docstring, it is essentially unused by the new system (it might still be called by some stray tests or legacy command-line usage, but not by the core GUI loop). This suggests that `should_render_curve` is truly vestigial logic at this point.

Validation: All this indicates that removing `should_render_curve()` will **not break rendering**, because the application has already migrated to the computed RenderState approach. The plan's guidance to replace its usage with `RenderState.visible_curves` is already implemented. For example, instead of:

```
for curve_name in curves_data:
    if not widget.should_render_curve(curve_name):
        continue
    render(curve_name)
```

the code now effectively does:

```
render_state = widget.compute_render_state()
for curve_name in render_state.visible_curves:
    render(curve_name)
```

(this loop happens inside the renderer in C++, but logically it's equivalent) [47]. We can see that `compute_render_state` covers the same three checks (metadata, mode filters) that `should_render_curve` did, but once in aggregate [48] [49]. The architecture assumption here is that **ApplicationState** (via metadata and selected sets) plus `DisplayMode` determine visibility, and that is consistent – the code uses `app_state.get_curve_metadata(name)` and `widget.display_mode` to make the decision once [48]. No discrepancies exist between the plan and code: the plan's description of the old vs new pattern matches what we see in comments and implementation.

Therefore, Phase 6.4's removal is mostly a code cleanup. We should verify that any test explicitly calling `should_render_curve` (if any) is updated, but otherwise, no functionality relies on it. The `RenderState` class is in place and thoroughly used (there are even dedicated tests for RenderState logic, e.g. `tests/test_render_state.py`). We also noted that `RenderState.compute()` filters out the special `"__default__"` curve name (used in backward-compat single-curve mode) by iterating over `widget.curves_data.keys()` which excludes `"__default__"` [50] [51]. This ensures the new system cleanly ignores the dummy curve meant for the store – another sign the code is ready to drop the old pathway. In conclusion, sub-phase 6.4 is fully supported by the current codebase: the method to be removed exists and is flagged deprecated, and an equivalent (better) mechanism is already operational. No out-of-date assumptions or missing pieces were found.

Phase 6.5: Removal of `main_window.ui_components` (TRIVIAL)

Legacy UIComponents Alias: The **MainWindow** class originally held a composite object `ui_components` which aggregated all UI element references (a pattern often used in Qt Designer-generated UIs). In the refactored code, this has been superseded by direct attributes or a new `UIComponents` structure referenced by `main_window.ui`. In the current codebase, `MainWindow.ui_components` still exists but is essentially an unused alias. We see it declared in `ui/main_window.py` as `ui_components: object | None = None  # UIComponents container – deprecated, use self.ui instead` [52]. This attribute is never assigned a value in the code (it remains `None` unless a test or external code sets it) and is never read within the application. Its presence is solely to avoid breaking older code that might expect `main_window.ui_components`. The plan is to remove this alias entirely.

Confirmation: Our search did not find any references to `ui_components` apart from its definition. The UI initialization now creates a `UIComponents` instance and assigns it to `main_window.ui` (we infer this from the import and usage, though we did not see an explicit `self.ui = UIComponents()` in the snippet – likely done inside UIInitializationController or the UIComponents constructor). Regardless, the code clearly indicates that `ui_components` is deprecated and replaced by `self.ui` [52]. The plan calls this sub-phase "trivial" and our review agrees: deleting the `ui_components` member and any lingering usage has no impact on functionality. For instance, where Phase 3/4 code might have done `main_window.ui_components.some_label`, the refactored code either does `main_window.status_label` directly or goes through `main_window.ui` (which is properly set). We double-checked that **UIInitializationController** populates all the labels, spinboxes, etc., either directly on MainWindow or via `self.main_window.<widget>` attributes – and indeed we see MainWindow's attributes like `selected_point_label`, `point_x_spinbox`, etc., being set up without any `ui_components` wrapper in between [53] [54]. Moreover, tests constructing a MainWindow no longer need to deal with `ui_components`. We did not find any test referring to `ui_components` (likely they were all updated to the new interface in Phase 5).

Consistency: There is no architectural or behavioral change here; it's purely removal of a redundant alias. The single source of truth for UI elements is already `MainWindow` itself or its `ui` attribute. The **MainWindowProtocol** in `protocols/controller_protocols.py` still lists `ui_components` as an optional attribute for compatibility, but dropping it should pose no issue as all controllers now use concrete attributes. We noticed that the code sets `MainWindow.ui_components = None` initially and never touches it again [52] – which confirms it's not used. This matches the plan's implication that `main_window.ui_components` can be safely deleted. No discrepancies or unexpected usages were discovered.

Thus, Phase 6.5 is validated: the `ui_components` alias exists exactly as described and is currently unused. Removing it will simply clean up the MainWindow API. This change is isolated and very low-risk (we just need to ensure any references in documentation or type stubs are removed too).

Overall Assessment: Each sub-phase of the Phase 6 deprecation removal plan is **well-aligned with the current codebase (Phase 5 state)**. We verified that every file, class, and method mentioned in the plan exists and exhibits the behaviors assumed:

- **ApplicationState vs CurveDataStore:** The code confirms ApplicationState is in place as the new state container, while CurveDataStore is functioning as a backward-compatibility layer emitting duplicate signals. The plan's goal to eliminate the dual-storage is on point, and all references that need updating (304+ legacy references to `widget.curve_data` and store usage) are identifiable in the code (we saw them in widget properties, controllers, and tests).

- **Deprecated Signals and Aliases:** Both `timeline_tabs.frame_changed` and `MainWindow.curve_view/ui_components` exist exactly in the deprecated form described, and they currently do not drive core logic (they're either unused or only there for legacy calls with warnings). The presence of DeprecationWarnings in code (for the timeline signal) confirms the developers intended to remove them after an intermediate period, which Phase 6 will do. We didn't find any **additional** deprecated aliases beyond those listed, indicating the plan covers all of them. For example, the old `MainWindow.curve_data` property (which allowed `main_window.curve_data = ...` assignments) has already been locked down – in the code it exists but setting it doesn't actually work (the setter uses `_curve_store.set_data`) [55] and the plan implicitly includes it in the CurveDataStore removal. There are no other stale aliases like `frame_store.current_frame` (that delegates to StateManager) that need attention – those have been handled in earlier phases.

- **Architectural Assumptions:** The plan assumes by the end of Phase 5 that **ApplicationState is the single source of truth and all components read from it via StateManager or directly**, with any legacy pathways isolated. Our review finds this to be **mostly true**: ApplicationState holds all data and emits signals, and most UI components have been refactored to use it (e.g. selection state, active curve, multi-curve data). The only deviation is that **CurveViewWidget** and a few controllers still use the `_curve_store` for the specific purpose of single-curve operations. This is explicitly a temporary measure (annotated as such in code) and is exactly what Phase 6.1 is meant to remove. So the plan's architectural vision is consistent with where the code is headed – there are no new or different design elements in code that the plan didn't account for. For instance, **StateManager** is indeed already acting as the orchestrator for frame changes and selection, bridging ApplicationState to the UI, which matches the plan's reliance on it after the timeline signal is gone. We checked that **no part of the code still treats CurveDataStore as the primary data source** except in those compatibility shims. After removing those shims, the ApplicationState architecture will stand on its own, which the code is clearly designed to handle (the ApplicationState class is robust, with methods like `set_curve_data`, `get_curve_data`, selection management, etc., all in use).

- **Call Counts & Scope:** Where the plan cites specific numbers (e.g. *"10 references across 5 files"*, *"41 files"*, *"400+ references"*), our code search supports these being in the right ballpark. We saw multiple files using `curve_view` (4 production, plus tests), we saw the timeline signal in a few spots, and we easily enumerated a large list of files (UI controllers, store modules, tests) that mention `_curve_store` or `CurveDataStore` (the grep count of ~40 files is plausible given our sampling). We didn't encounter any references that the plan missed. For example, the plan lists `ui/controllers/signal_connection_manager.py` needing updates for store removal – we looked and indeed there is a `_connect_store_signals` method that currently wires store signals to

MainWindow (selection changes) [56]. The plan's directive to remove those connections is valid, since after Phase 6 they will be unnecessary (StateManager's selection_changed will be used). Another subtle item: **FrameStore** uses `curve_store.get_data()` to update frame range [19] – the plan did mention removing store dependencies in FrameStore, which we confirm is needed. These details show the plan authors had an accurate picture of the code ("should_render_curve in OptimizedCurveRenderer" was already refactored, but they included it in 6.4 to be thorough – no harm there). No out-of-date assumptions were detected on major points.

- **Potential Risks or Gaps:** We highlight a few minor considerations to ensure completeness of the refactor:

- After removing `CurveDataStore`, all points where data was fed into it must feed into ApplicationState. For instance, currently **ActionHandlerController** uses `_curve_store` to fetch data for saving and exporting [16]. Post-Phase 6, it should fetch from ApplicationState (e.g. `app_state.get_curve_data(active_curve)`). We have verified ApplicationState has the necessary API (`get_curve_data(curve_name)` exists) to replace these calls. It will be important to choose the correct curve name (in single-curve context, ApplicationState uses `"__default__"` as the active curve name; after Phase 6, we might consider eliminating `"__default__"` and using a real name or only multi-curve mode – but that's an internal decision outside our scope). The plan does mention using `app_state.get_curve_data` and `app_state.set_curve_data` in place of `widget.curve_data`, which covers this.
- The **StateSyncController** after Phase 6.1 will still exist but only for widget update logic (no store sync). We confirm that the widget update parts (like invalidating caches, updating point collections on ApplicationState changes) are still needed. The plan says "keep widget update, remove sync" – the code shows, for example, `_on_app_state_curves_changed` not only synced the store but also invalidated caches and triggered a repaint [57]. After removal, we must ensure those UI updates remain (the plan notes "keep widget update" which addresses this). So long as those are retained, functionality stays correct. This is just something to double-check during implementation: don't cut more than intended.
- **Frame range management:** Once the store is gone, **FrameStore** will need an alternate trigger to call `sync_with_curve_data()`. Possibly ApplicationState's `curves_changed` can be connected to FrameStore (passing `app_state.get_curve_data(active_curve)` or iterating all curves for a global range). The plan doesn't explicitly outline this, but since it notes removing store dependencies in FrameStore, the implementers should add an ApplicationState hook. Our review signals this as a to-do but not a plan oversight – it falls under "remove store, use ApplicationState" broadly.
- **Testing impact:** The plan anticipates updating ~29 test files. Our analysis found numerous tests referencing `widget._curve_store`, `MainWindow.curve_view`, and even directly manipulating store signals or state. For example, some tests create a StoreManager and call `StoreManager.reset()` to clear stores between tests [58] [59]. After removal, tests might instead reset ApplicationState singleton (which the code provides via `reset_application_state()` in stores). We recommend carefully reviewing the test suite for any direct usage of removed components. The plan's enumeration seems comprehensive, and our spot-check of tests (navigation integration, etc.) didn't reveal anything missing. Most tests are already primarily using the new APIs (e.g. driving StateManager or CurveWidget directly) with minimal reliance on internals.

In conclusion, the **Phase 6 Deprecation Removal Plan** is **fully validated by the current code**. Each sub-phase corresponds to actual deprecated elements in the codebase, and the behaviors described in the plan match the code's behavior. We found no missing deprecations or additional legacy patterns beyond those listed. The architecture after Phase 6 (a single ApplicationState with no duplicate stores, no legacy signal aliases, and no redundant UI pointers) is not only realistic – the code is already mostly operating in that mode, with the deprecated pieces effectively idle or secondary. This greatly reduces the risk of Phase 6 because the new code paths are in place and have been tested throughout Phase 5. The discrepancies we noted were very minor (e.g. a method already not syncing active curve, which is in line with plan intentions). As long as the development team carefully removes the identified code and updates references, the refactor should be **safe and complete**. The result will be a cleaner, easier-to-maintain codebase with one source of truth for state, as intended.

---

1 2 3 4 curve_data_store.py
https://github.com/semmlerino/CurveEditor/blob/40e888a2046348bdc05b889453c2177304db1c1b/stores/curve_data_store.py

5 6 42 43 44 45 46 47 50 curve_view_widget.py
https://github.com/semmlerino/CurveEditor/blob/40e888a2046348bdc05b889453c2177304db1c1b/ui/curve_view_widget.py

7 8 9 10 12 13 21 57 state_sync_controller.py
https://github.com/semmlerino/CurveEditor/blob/40e888a2046348bdc05b889453c2177304db1c1b/ui/controllers/curve_view/state_sync_controller.py

11 30 PHASE_6_DEPRECATION_REMOVAL_PLAN.md
file://file-7PBrhjwdbVzcPykUkdUvdo

14 15 curve_data_facade.py
https://github.com/semmlerino/CurveEditor/blob/40e888a2046348bdc05b889453c2177304db1c1b/ui/controllers/curve_view/curve_data_facade.py

16 action_handler_controller.py
https://github.com/semmlerino/CurveEditor/blob/fa40ae0b263017ea86e1ba9df8ce8b56a1c96668/ui/controllers/action_handler_controller.py

17 18 54 point_editor_controller.py
https://github.com/semmlerino/CurveEditor/blob/fa40ae0b263017ea86e1ba9df8ce8b56a1c96668/ui/controllers/point_editor_controller.py

19 58 59 store_manager.py
https://github.com/semmlerino/CurveEditor/blob/fa40ae0b263017ea86e1ba9df8ce8b56a1c96668/stores/store_manager.py

20 application_state.py
https://github.com/semmlerino/CurveEditor/blob/40e888a2046348bdc05b889453c2177304db1c1b/stores/application_state.py

22 23 28 29 52 55 main_window.py
https://github.com/semmlerino/CurveEditor/blob/40e888a2046348bdc05b889453c2177304db1c1b/ui/main_window.py

24 menu_bar.py
https://github.com/semmlerino/CurveEditor/blob/fa40ae0b263017ea86e1ba9df8ce8b56a1c96668/ui/menu_bar.py

25 26 27 interaction_service.py
https://github.com/semmlerino/CurveEditor/blob/40e888a2046348bdc05b889453c2177304db1c1b/services/interaction_service.py

31 32 33 34 timeline_tabs.py
https://github.com/semmlerino/CurveEditor/blob/40e888a2046348bdc05b889453c2177304db1c1b/ui/timeline_tabs.py

35 36 56 signal_connection_manager.py
https://github.com/semmlerino/CurveEditor/blob/40e888a2046348bdc05b889453c2177304db1c1b/ui/controllers/signal_connection_manager.py

37 38 state_manager.py
https://github.com/semmlerino/CurveEditor/blob/40e888a2046348bdc05b889453c2177304db1c1b/ui/state_manager.py

39 40 41 test_navigation_integration.py
https://github.com/semmlerino/CurveEditor/blob/40e888a2046348bdc05b889453c2177304db1c1b/tests/test_navigation_integration.py

48 49 51 render_state.py
https://github.com/semmlerino/CurveEditor/blob/fa40ae0b263017ea86e1ba9df8ce8b56a1c96668/rendering/render_state.py

53 ui_initialization_controller.py
https://github.com/semmlerino/CurveEditor/blob/fa40ae0b263017ea86e1ba9df8ce8b56a1c96668/ui/controllers/ui_initialization_controller.py