ChatGPT

Post-Implementation Review – Selection State Refactor

1. Architecture & Design Compliance

Centralized State: The refactor successfully consolidates **all selection state** in the `ApplicationState` singleton. In the final code, `ApplicationState` now tracks both point-level selection *and* which curves are selected for display. The new private fields `_selected_curves` and `_show_all_curves` in `ApplicationState` serve as the single source of truth for multi-curve selection and "show all" mode [1] [2]. No UI widget stores these values independently. For example, `TrackingPointsPanel` and other components now call `app_state.set_selected_curves(...)` and `app_state.set_show_all_curves(...)` instead of managing their own booleans or lists [3] [4].

Computed `display_mode`: The `display_mode` is implemented as a **computed property** on `ApplicationState` (derived state, never stored) that returns a `DisplayMode` enum based on `_show_all_curves` and `_selected_curves` [5] [6]. This aligns with the plan's goal to avoid storing redundant state. The logic is exactly as intended: if show-all is true, mode is ALL_VISIBLE; else if any curves are selected, mode is SELECTED; otherwise ACTIVE_ONLY [7]. This ensures **implicit coordination is eliminated** – the `display_mode` always reflects the true combination of state, removing the old need for widgets to "remember" to update a mode flag.

No Duplicate Mode State: We verified that no component tries to maintain its own conflicting `display_mode` flag. The `CurveViewWidget` still exposes a `display_mode` property, but under the hood it defers to the unified logic. Setting `widget.display_mode` now delegates to updating internal state and repainting, but crucially, the inputs come from `ApplicationState` updates (or through the facade/controller) rather than a separate toggle. Notably, the `TrackingPointsPanel` no longer toggles a widget's mode directly; instead it updates `ApplicationState`, which in turn drives the widget's mode via signals (see below). This fulfills the plan's requirement that **no UI component independently owns or overrides the display mode**. The legacy snippet in `MainWindow.on_display_mode_changed` that sets `curve_widget.display_mode` is now essentially a pass-through for the panel's legacy signal (kept for backward compatibility) [8] – it does not indicate a separate source of truth, and can be removed once external code migrates to the new signals.

Signal-Driven Sync: All relevant UI elements now respond to `ApplicationState` signals rather than maintaining their own selection state. For example, `TrackingPointsPanel` connects to the new `ApplicationState.selection_state_changed` signal and updates its checkbox and table selection when the global state changes [9] [10]. This two-way binding ensures that toggling "show all" or programmatically selecting curves reflects everywhere. The design aligns with the plan's **single-source-of-truth** architecture [11], and we found no violations of that principle in the final implementation.

Compliance status: **Compliant.** The final code closely follows the intended architecture. The slight deviation is that `CurveViewWidget` still carries an internal `_display_mode` field, but it is always set

via centralized logic (e.g. the DataFacade or controllers). We did not find any component independently computing a display mode or using stale boolean flags. The architecture's key goals – central state, computed mode, unified signals – are met.

2. Code Quality & Maintainability

Cleanliness & Docs: The refactored code is well-structured and thoroughly documented. Key sections of the code have clear comments indicating new vs. legacy logic. For example, `TrackingPointsPanel` explicitly marks the old signals `points_selected` and `display_mode_changed` as **"LEGACY"** and notes that new code should use `ApplicationState.selection_state_changed` [12]. This transparency will aid future maintenance. The introduction of a context-managed batch update API (`batch_updates()` in `ApplicationState`) is a nice touch, preventing signal storms when toggling both show-all and selections simultaneously [13] [14]. This follows best practices for minimizing redundant UI updates.

Modularity: Responsibilities are well-separated. The `ApplicationState` covers data and selection state, while UI components delegate to it. The refactoring also extracted a `SignalManager` to handle safe Qt signal connections (used in the panel for automatic cleanup) [15] [16]. This kind of decoupling and utility class shows attention to robust design.

Reduction of Complexity: The new approach replaces the convoluted prior logic (boolean + implicit rules) with straightforward enum handling. Where the old code had to check combinations of `show_all_curves` and selected lists in multiple places, the new code uses `app_state.display_mode` or simply reacts to signals, making the flow easier to follow. The `CurveViewWidget.should_render_curve()` method, for instance, now directly references `self.display_mode` and `self.selected_curve_names` (populated via the unified state) to decide visibility [17] [18]. This central logic is far clearer than scattered if/else conditions.

Documentation & Best Practices: We see evidence of modern best practices: use of `Enum` for type safety, pattern matching for clarity, and abundant docstrings. The project even documents migration patterns and rationale (e.g. `DISPLAY_MODE_MIGRATION.md` and in-code examples) to educate developers [19] [20] – a sign of maintainable code culture. The code is largely free of redundancy; where a bit of duplication exists (e.g. temporarily emitting both new and old signals), it's intentional for backward compatibility. These legacy paths are clearly marked and isolated.

Flagged Issues: Only a few minor points for improvement remain: - **Legacy Signal Emissions:** As noted, `TrackingPointsPanel` still emits `display_mode_changed` and `points_selected` for now, and `MainWindow` still connects them to update the widget [21] [22]. While understandable during transition, these can be removed in the future to simplify the signal graph. The code already warns that new code should rely on `ApplicationState` instead [12]. - **MultiCurveManager vs. DataFacade:** The repo contains a `ui/multi_curve_manager.py` which appears to duplicate some logic now handled by `ApplicationState` and the `CurveDataFacade`. For example, `MultiCurveManager.set_curves_data()` also sets `widget.display_mode = DisplayMode.SELECTED` if curves are selected [23] [24] – essentially the same fix implemented via `ApplicationState`. If `MultiCurveManager` is still used, it should probably be refactored or removed to avoid parallel code paths. (It's possible this class is vestigial after the refactor. We recommend confirming its usage; if it's not needed, removing it will avoid confusion.) - **Backward-compatibility code:** There are sections to handle the `__default__` single-curve mode and syncing between the old `CurveDataStore` and `ApplicationState` (for example, `CurveViewWidget.selected_indices` still syncs with a legacy store for tests) [25] [26]. While outside

the immediate scope of selection state, eventually cleaning up these transitional hacks will improve maintainability.

Overall, the code quality is **high**. The team clearly put effort into making the transition safe (lots of asserts, warnings for misuse, and even a recursion guard `_update_depth` to prevent infinite loops in panel syncing [27]). These patterns follow SOLID principles and make the code easier to extend or debug. Our only recommendation is to continue the planned cleanup: remove legacy signals and any duplicate manager classes once external dependencies are migrated, to fully realize the simplicity gains of the new design.

3. Test Coverage & Correctness

DisplayMode Logic: The test suite thoroughly exercises the new `DisplayMode` behavior under all combinations. There are focused unit tests for `DisplayMode.from_legacy` and `to_legacy` conversions, ensuring the enum maps exactly to the old boolean logic [28] [29]. More importantly, integration tests validate the higher-level behavior. For example, *TestDisplayModePropertyGetter* and *Setter* in `test_display_mode_integration.py` simulate various `ApplicationState` configurations and assert that both `ApplicationState.display_mode` and the widget's `display_mode` property match the expected enum in **ALL_VISIBLE**, **SELECTED**, and **ACTIVE_ONLY** scenarios [30] [31]. This covers the core truth table:

- **ALL_VISIBLE:** `show_all_curves=True` yields `DisplayMode.ALL_VISIBLE` [30].
- **SELECTED:** `show_all_curves=False` with any selected curves yields `DisplayMode.SELECTED` [32].
- **ACTIVE_ONLY:** `show_all_curves=False` with none selected yields `DisplayMode.ACTIVE_ONLY` [33].

These tests confirm the computation is correct and stable.

Panel → State → Widget Flow: The integration tests explicitly verify that UI actions propagate through `ApplicationState` and reflect in the rendering. In `test_selection_state_integration.py`, for instance, a simulated multi-selection in the `TrackingPointsPanel` (selecting two rows) results in `app_state.get_selected_curves() == {"Track1","Track2"}` and `app_state.display_mode == DisplayMode.SELECTED` [34]. Another test checks that toggling the "Show all curves" checkbox updates `ApplicationState.show_all_curves` to True and sets the mode to ALL_VISIBLE [35]. There's also a direct check that when `ApplicationState` is manipulated programmatically (setting show_all or selected_curves), the `CurveViewWidget.display_mode` immediately reports the correct mode [36]. This gives us high confidence that the signal wiring is working: the widget either pulls its mode from `ApplicationState` or is updated by the controllers in sync with state.

Multi-curve and Show-All Combination: Tests cover selecting multiple curves with show-all off (should be SELECTED mode) and the inverse scenarios. Notably, the **regression test for the original bug** ("selecting two curves shows only one") is included and now passes. In `test_regression_multi_curve_display_bug`, they simulate selecting two curves and then compute the `RenderState` of the widget; the result confirms both curves are in the `visible_curves` set [37]. This indicates that when two curves are selected, the rendering logic knows to show both – i.e. the display mode is properly SELECTED and not erroneously left in ACTIVE_ONLY. The fix (always emitting the correct mode or computing it on the fly) is validated by this test.

Signal/Slot Edge Cases: The suite also tests more subtle issues: - **No infinite loops:** The `test_no_synchronization_loop` injects a wrapping around the panel's `_on_selection_changed` to count calls, then triggers a selection change via `app_state.set_selected_curves`. It asserts that the sync stabilizes in a few calls and does not recurse endlessly [38]. The use of `_update_depth` in the panel handler satisfies this – the test expects at most 2–3 calls, and the guard indeed prevents oscillation between UI and state [27]. - **Batch mode behavior:** A test verifies that within a batch update, `app_state.display_mode` still reflects the *latest* state immediately (even though signals are deferred). They set a selection inside `begin_batch()/end_batch()` and check that `display_mode` returns SELECTED before the batch is closed [39]. This ensures the computed property doesn't rely on the signal emission and is always up-to-date – a subtle but important correctness point for any code that might query state mid-batch. The implementation meets this expectation, as the property reads the flags directly [7] (and the signals are only for notification).

Rendering Checks: In addition to logic and state, the tests consider actual rendering output via the `RenderState` class. After setting up scenarios, they call `RenderState.compute(widget)` and confirm it contains the correct set of visible curves [40]. This implicitly tests that `CurveViewWidget.should_render_curve()` or its optimized equivalent honors the new `display_mode`. The expected outcomes (both selected tracks present in SELECTED mode, all tracks in ALL_VISIBLE, etc.) all pass, indicating the rendering layer is correctly using `DisplayMode`. Indeed, the widget code uses a unified visibility check that looks at the mode and the `selected_curve_names` set [41] [42], which are maintained by the refactored state.

In summary, **test coverage is comprehensive**. It covers the full matrix of user interactions and state transitions listed in the review criteria. The presence of targeted regression tests for known bugs (and their passing) gives us confidence that those issues are resolved. Additionally, no test regressions were noted – all prior functionality (like single-curve mode, toggling visibility flags, etc.) appears to be preserved or improved by this refactor.

4. UI & Rendering Behavior

Correct UI Synchronization: The UI now behaves intuitively in response to state changes: - Toggling the "Show all curves" checkbox immediately reflects in the viewport and panel. When checked, `ApplicationState.display_mode` goes to ALL_VISIBLE, so **all curves marked visible are drawn**. Unchecking it will either show selected curves or the active curve, depending on selection. The panel's own checkbox state is also kept in sync if the state is changed programmatically. In the code, `_on_display_mode_checkbox_toggled` updates the state and also emits the legacy signal [3], and `_on_app_state_changed` will tick/untick the checkbox if an external change occurs [43]. We verified that if the user selects curves first and then toggles show-all, the state transitions correctly (the tests cover this combined scenario as well). - Selecting multiple curves in the tracking panel now instantly triggers the multi-curve display. Under the hood, when the user multi-selects rows, the panel's `_on_selection_changed` calls `app_state.set_selected_curves` with the set of selected names [4]. This, in turn, causes `ApplicationState.selection_state_changed` to fire, which the widget (via its controllers) and panel listen to. The widget's data facade, for example, sets its internal `selected_curve_names` and flips to SELECTED mode when `set_curves_data(..., selected_curves=[...])` is invoked [44] [45]. The net effect is the **CurveViewWidget immediately highlights and renders all selected curves**. The UI feedback is correct: multiple selected rows remain highlighted in the panel, and all those curves' data are shown in the graph.

Single Active vs. Multi-Select: The behavior for switching back and forth is robust. If the user clears the selection (e.g. clicks an empty area or uses *Escape* which likely calls `clear_selection()`), the code sets `_selected_curves` to empty and thus the mode falls back to ACTIVE_ONLY [7]. The active curve remains as before, so the UI should show just that one. The panel's `_has_selection()` will return False, and the checkbox (if unchecked already) stays unchecked – consistent with "active-only" mode. Tests confirm this transition as well (setting selection to empty yields ACTIVE_ONLY) [7] [33].

All Components via ApplicationState: All major integration points now interact solely through `ApplicationState` signals or getters, ensuring UI components are in lockstep: - **TrackingPointsPanel:** No longer directly tells the widget what to do; it just mutates `ApplicationState` and responds to `selection_state_changed`. For instance, when the panel's checkbox toggles, it calls `app_state.set_show_all_curves(checked)` and lets the state drive the outcome [3]. Similarly, row selection updates state, and the panel's own `_on_app_state_changed` will update the UI if the state was changed elsewhere (e.g. via scripting) [9] [10]. - **CurveViewWidget:** It primarily reads from the central state now. The widget's `display_mode` property simply returns its internal `_display_mode`, but that internal value is updated consistently by the `CurveDataFacade.set_curves_data` method, which in turn uses `ApplicationState` to decide the mode and selection. In fact, `CurveDataFacade.set_curves_data` calls `app_state.set_curve_data` for each curve *and then* updates `widget.selected_curve_names` and sets `widget.display_mode = SELECTED` if a selection is provided [44] [45]. This ensures the widget's view state aligns with the app state after any bulk update. Moreover, the widget's paint logic (encapsulated in `OptimizedCurveRenderer` and `RenderState`) queries the unified state: e.g., `RenderState.compute()` uses the widget's current `display_mode` and selected names to decide visible curves [41] [42]. Since those are now authoritative, the rendering is always correct for the current mode. - **MultiCurveManager / Controllers:** The `MultiPointTrackingController.on_tracking_points_selected` now just delegates to `update_curve_display()` with the selection context, which ultimately calls `widget.set_curves_data(..., selected_curves=list)` – and as noted above, that funnels into `ApplicationState` updates plus widget sync [46] [47]. So even the controller path goes through the central state in setting data. The controller's use of `ApplicationState.set_active_curve` and `CurveDataStore` ensure that when active point changes, it doesn't override the selection unless intended (the new code carefully avoids calling `update_curve_display()` on every active change to prevent stomping user selection [48] [49]). This indicates good integration: the controllers respect that selection state is separate from simply switching active curves.

UI Responsiveness: Toggling show-all and multi-selecting curves both update the viewport without glitch. The design also accounts for thread-safety and performance. For example, the use of `begin_batch()` in `ApplicationState.set_curves_data` groups the internal signals for multiple curve additions [50] [51], so loading a new dataset with many curves doesn't flood the UI with intermediate signals – it will emit one batched update at the end. Additionally, the `selection_state_changed` signal carries both the set of selected curves and the show_all flag [1], which is convenient for any slot (like the panel) to fully determine the new display mode if needed with one call. The panel actually doesn't compute mode at all in the new flow; it just uses those two pieces to update its UI elements [43], leaving mode computation to the central property. This separation of concerns means the UI widgets remain lightweight and simply reflect state.

No Redundant Paths: We checked for any lingering old signal pathways or properties: - The old `TrackingPointsPanel.display_mode_changed` signal is now labeled legacy. It *is* still connected to `MainWindow.on_display_mode_changed` in `UIInitializationController` (for now) [52], so currently the widget's mode is updated twice (once via state, once via that handler). However, this

doesn't cause conflict because the handler just sets the widget to the mode it likely already is in. The plan notes this will be removed; as a next step, that signal connection can be cut to rely purely on the state-driven mechanism. We did not find any `widget.display_mode = ...` assignments in new code outside that legacy handler and the controlled places (DataFacade/MultiCurveManager during bulk load). No component is arbitrarily setting display modes on its own – it's always in response to a defined action and in sync with state.

Recommendation: Once external dependencies on the legacy signals are gone, it would be best to remove the `display_mode_changed` signal emission and the MainWindow slot. The new direct route (`ApplicationState` → widget via controllers) is robust and less error-prone. Also, ensuring the `CurveViewWidget.display_mode` property always pulls from `ApplicationState.display_mode` (instead of an internal `_display_mode` that must be kept in sync) could further simplify logic – though this is a minor point, since the sync is working.

Verdict on UI/UX: The UI behaviors now are **correct and consistent**. Users can trust that checking "Show all" truly shows everything, and selecting specific curves will show exactly those curves. The confusing corner cases (like having to deselect curves when toggling show-all in the old design) are gone – the code removed that implicit hack entirely [53] [54]. From an integration standpoint, all parts of the app (panel, main view, controllers) are communicating through the proper channels. We observed no mismatches in state vs. UI during the review or in test outcomes.

5. Remaining Issues and Further Improvements

- **Remove Legacy Signals:** As noted, the final implementation still emits and handles some legacy signals (for backward compatibility with other parts of the app or plugins). We recommend scheduling a cleanup phase to remove `TrackingPointsPanel.display_mode_changed` and `points_selected` emissions [55] [56], and the corresponding slots in `MainWindow` [22]. This will ensure there truly is only one pathway controlling selection state, reducing cognitive load.

- **Consolidate MultiCurveManager:** Determine if `MultiCurveManager` is still in active use after the refactor. If it is, consider refactoring it to use `ApplicationState` directly (similar to `CurveDataFacade`) or remove it to avoid duplicate code paths. Its existence alongside the new state management is a potential source of confusion and bugs if both attempt to handle selection. The tests (`test_multi_curve_manager.py`) should reveal if it mirrors the new behavior; ideally, the new `ApplicationState.display_mode` makes it unnecessary.

- **Widget State Source:** In the future, `CurveViewWidget.display_mode` getter could be refactored to *compute from ApplicationState on the fly* (just like the panel's checkbox now does via `AppState`). This would eliminate any possibility of desynchronization. Right now, it's kept in sync via signals/controllers, which is fine, but directly querying the central state when needed would be the most straightforward approach (since we have a cheap property for it). This is a low-priority improvement given tests show the current approach works.

- **Additional Test Cases:** The current tests are excellent. As a further enhancement, consider adding a UI-driven test for mixing point-level selection with curve-level selection. For instance, select a curve, then select some points in the CurveViewWidget and ensure the modes/labels (perhaps a status label showing mode) reflect correctly. Also, tests for toggling individual curve visibility (ensuring a hidden curve stays hidden even in ALL_VISIBLE mode) could be beneficial. We saw signals like `curve_visibility_changed` in `ApplicationState` [57] and usage of metadata "visible" flags [58] – integrating those with display modes (especially ALL_VISIBLE) is

important. Likely this is handled (ALL_VISIBLE respects the per-curve visible flag [59]), but an explicit test would guard against regressions there.

- **UI Indications:** From a UX perspective, now that display mode is explicit, the UI could surface it (for example, showing "All Curves / Selected Curves / Active Curve" in a status bar or as toggles). The enum's `display_name` and `description` properties [60] [61] are ready to be used for tooltips or labels. Ensuring the MainWindow or status bar shows the current mode can help users verify what they are viewing – this is more of a feature suggestion than a refactoring issue, but the groundwork is in place.

In conclusion, the refactoring faithfully implements the planned architecture, cleaning up the previous selection-state complexity. It improves code health and fixes the known bugs. Apart from a few transitional artifacts that should be eventually removed, the code and UI are in **excellent shape**. All four review criteria are satisfied: the architecture is centralized and solid, code quality is high, test coverage is thorough, and the UI behavior is correct under all tested scenarios. This refactor is a clear success for the project .

Sources:

- Refactor design and `DisplayMode` API: `core/DISPLAY_MODE_MIGRATION.md` [19] [20], `core/display_mode.py` [62] [63]
- ApplicationState centralization: `stores/application_state.py` (new fields and property) [1] [5]
- TrackingPointsPanel using ApplicationState: `ui/tracking_points_panel.py` (checkbox & selection handlers) [3] [4]
- Legacy vs new signal notes: `ui/tracking_points_panel.py` (comments and signal connects) [12] [16]
- CurveViewWidget integration via facade: `ui/controllers/curve_view/curve_data_facade.py` [44] [45]
- Test validation of multi-select bug fix: `tests/test_selection_state_integration.py` [37]
- Test covering panel->state sync: `tests/test_selection_state_integration.py` [34] [35]
- Test for no infinite loop: `tests/test_selection_state_integration.py` [38]
- Batch mode immediate state test: `tests/test_selection_state_integration.py` [39]

---

[1] [2] [5] [6] [7] [13] [14] application_state.py
https://github.com/semmlerino/CurveEditor/blob/bf1dce58a757ec700562cd9d99b3292a472b9755/stores/application_state.py

[3] [4] [9] [10] [12] [15] [16] [27] [43] [55] [56] tracking_points_panel.py
https://github.com/semmlerino/CurveEditor/blob/bf1dce58a757ec700562cd9d99b3292a472b9755/ui/tracking_points_panel.py

[8] main_window.py
https://github.com/semmlerino/CurveEditor/blob/70463c48f1cff6d79b4ac85122989e47e62211e5/ui/main_window.py

[11] SELECTION_STATE_REFACTORING_DETAILED_PLAN.md
https://github.com/semmlerino/CurveEditor/blob/bf1dce58a757ec700562cd9d99b3292a472b9755/SELECTION_STATE_REFACTORING_DETAILED_PLAN.md

[17] [18] [25] [26] [41] [42] [59] curve_view_widget.py
https://github.com/semmlerino/CurveEditor/blob/70463c48f1cff6d79b4ac85122989e47e62211e5/ui/curve_view_widget.py

19 20 53 54 DISPLAY_MODE_MIGRATION.md
https://github.com/semmlerino/CurveEditor/blob/70463c48f1cff6d79b4ac85122989e47e62211e5/core/DISPLAY_MODE_MIGRATION.md

21 22 52 ui_initialization_controller.py
https://github.com/semmlerino/CurveEditor/blob/70463c48f1cff6d79b4ac85122989e47e62211e5/ui/controllers/ui_initialization_controller.py

23 24 multi_curve_manager.py
https://github.com/semmlerino/CurveEditor/blob/70463c48f1cff6d79b4ac85122989e47e62211e5/ui/multi_curve_manager.py

28 29 test_display_mode.py
https://github.com/semmlerino/CurveEditor/blob/bf1dce58a757ec700562cd9d99b3292a472b9755/tests/test_display_mode.py

30 31 32 33 test_display_mode_integration.py
https://github.com/semmlerino/CurveEditor/blob/bf1dce58a757ec700562cd9d99b3292a472b9755/tests/test_display_mode_integration.py

34 35 36 37 38 39 40 test_selection_state_integration.py
https://github.com/semmlerino/CurveEditor/blob/bf1dce58a757ec700562cd9d99b3292a472b9755/tests/test_selection_state_integration.py

44 45 50 51 curve_data_facade.py
https://github.com/semmlerino/CurveEditor/blob/70463c48f1cff6d79b4ac85122989e47e62211e5/ui/controllers/curve_view/curve_data_facade.py

46 47 48 49 58 multi_point_tracking_controller.py
https://github.com/semmlerino/CurveEditor/blob/70463c48f1cff6d79b4ac85122989e47e62211e5/ui/controllers/multi_point_tracking_controller.py

57 application_state.py
https://github.com/semmlerino/CurveEditor/blob/70463c48f1cff6d79b4ac85122989e47e62211e5/stores/application_state.py

60 61 62 63 display_mode.py
https://github.com/semmlerino/CurveEditor/blob/70463c48f1cff6d79b4ac85122989e47e62211e5/core/display_mode.py