ChatGPT

Audit Report: StateManager Migration Plan vs. Current Code

File & Class Existence

- `stores/application_state.py` – **Exists.** Defines the `ApplicationState` class as described [1]. This file contains the central application state logic.
- `ui/state_manager.py` – **Exists.** Defines the `StateManager` class for UI-related state [2]. This corresponds to the UI preferences layer in the plan.
- **Other referenced files:** The plan mentions a `data/data_operations.py` (for file I/O) which is **not present**. Instead, file loading/saving is handled by `services/data_service.py` in the current codebase [3] [4]. References in the plan to `data_operations.py` should be mapped to `data_service.py`. Files like `ui/main_window.py` and various controllers exist and interface with `StateManager` and `ApplicationState` as expected.

ApplicationState vs StateManager: Properties & Migration Targets

- `ApplicationState` **class:** Present and aligns with the data-layer role. It holds curve data and related state (multi-curve support) as per design [1]. Notably, `ApplicationState.active_curve` is implemented as a **property** (with getter/setter) – there is no `get_active_curve_name()` method. This confirms the plan's correction that code uses a property instead of a method [5]. All code references to the active curve use the property, so the plan's instruction to use `self.active_curve` is already in effect.
- **Stateful properties currently in StateManager:** The following properties identified for migration are indeed **present in** `StateManager` **and not in** `ApplicationState`**:**
- `track_data`: stored in `StateManager` (`_track_data` list) with getters/setters [6]. **Not** present in `ApplicationState` (no `track_data` attribute or single-curve storage there yet). ⚠
- `image_files` and `image_directory`: maintained in `StateManager` (`_image_files` list and `_image_directory` string) [7] [8]. **Not** in `ApplicationState` currently – the plan calls for adding these to the data layer in Phase 2. ⚠
- `total_frames`: tracked in `StateManager` (`_total_frames` int with property) [9] [10]. **Not** in `ApplicationState` yet (plan intends to migrate this as it's derived from image_files). ⚠
- These findings validate the plan's identification of **improperly located data**. Curve data (`track_data`), image sequence, and frame count are still in `StateManager`, confirming the need to migrate them to `ApplicationState` (data layer). The code currently uses these in `StateManager`, e.g. setting image files in the UI layer updates `_image_files` and `_total_frames` there [7] [10].
- **Properties correctly in StateManager:** The plan's list of UI/session state in `StateManager` is accurate. Properties like `zoom_level`, `pan_offset`, `view_bounds`, `current_tool`, `hover_point`, smoothing settings, `window_position`, `splitter_sizes`, `is_fullscreen`, `recent_directories`, etc., all exist in `StateManager` [11]. For example, `current_tool` is stored as `_current_tool` with a property getter/setter [12]. These are

indeed UI state, not duplicated in `ApplicationState` (which focuses on curve and frame data), matching the plan.

- **Signals for these:** Many UI properties have no signals (by design). E.g. `hover_point` setter does not emit a signal (per plan, it's intentional for performance) [13]. This is consistent with the plan's notes.
- `active_curve` **handling:** In `ApplicationState`, `active_curve` is a property that on change emits `active_curve_changed` [14]. The plan's Phase 0.1 noted a mistaken reference to `get_active_curve_name()`. We confirm the code uses the property correctly; there is no such method in the code (so the plan's correction is already reflected in code usage).
- **Legacy single-curve methods:** The plan proposes adding `ApplicationState.set_track_data/get_track_data/has_data` for compatibility. **These do not exist yet** in the code (no `set_track_data` method in `ApplicationState` as of now) [15]. This is a planned addition. ⚠ Currently, `StateManager` provides `set_track_data(...)` which operates on its internal `_track_data` [16]. After migration, this will delegate to `ApplicationState` (as outlined in the plan [17]), but that change is not in the code yet. We will need to implement those new methods and refactor `StateManager.set_track_data` accordingly.

Signal Architecture & Thread-Safety

- **Single source of truth for signals:** The plan calls for eliminating duplicate signals. In the current code, **there is no** `StateManager.track_data_changed` **signal at all** – only `ApplicationState.curves_changed` signifies changes in curve data. This means the duplicate signal issue has effectively been addressed already (the plan's mention of `StateManager.track_data_changed` appears to reference an older design) [18]. All curve data change notifications come from `ApplicationState.curves_changed`. This aligns with *Principle 2: Single Signal Source per Data Type* – we observed no conflicting signals for curve data in the code.
- **Existing signals in** `ApplicationState`**:** The code defines signals for core data changes: e.g. `curves_changed`, `selection_changed`, `active_curve_changed`, `frame_changed`, etc. [19]. Notably, `ApplicationState.curves_changed` **in code emits a** `dict` **of all curves** (Signal(dict)) [20], whereas the plan describes it as emitting a curve name (Signal(str)) [21]. This is a **discrepancy** – the implementation has evolved to send a dictionary of all curves (to support multi-curve workflows), so the plan's documentation of that signal is slightly outdated. ⚠ Migration steps should take into account that any connected slots expect a `dict` from `curves_changed` (or adjust if needed).
- **Existing signals in** `StateManager`**:** Currently includes signals for file changes, modification status, view changes, selection (point indices), frame changes, playback state, and active timeline point [22]. We confirm these match the plan's notion of StateManager as a UI layer:
- `file_changed`, `modified_changed` – yes, in code [23] (plan marks these as correct UI signals).
- `view_state_changed` – yes, exists [24]. However, in code this signal is emitted in limited cases (e.g., on reset) and not every time a view property changes – the plan suggests it should be the unified view-state signal. We see that e.g. `zoom_level` and `pan_offset` setters in `StateManager` currently just log changes without emitting `view_state_changed` [25] [26]. This indicates an **implementation gap**: the plan calls for using `view_state_changed` whenever view parameters change, which is not fully done yet (likely to be addressed in migration). ⚠
- **No undo/redo/tool signals yet:** As expected, `StateManager` **lacks** `undo_state_changed`, `redo_state_changed`, and `tool_state_changed` signals. The code defines properties

`can_undo`, `can_redo`, and `current_tool` but changes to them do not emit any signal [27] [26]. This confirms the plan's note that these signals need to be added. ⚠

- **Thread-safety – ApplicationState:** The code adheres to the thread-safety model described in the plan:
- All state mutations in `ApplicationState` call `self._assert_main_thread()` at the start to enforce main-thread only access [14] [28]. If any background thread tried to call these, an assertion would fire. This matches the plan's requirement that `ApplicationState` is main-thread only.
- `ApplicationState` uses an internal QMutex (`self._mutex`) exclusively to guard the batch-update mode flag and pending signals list [29]. There are **no direct calls to `_mutex.lock()` or `unlock()` in the code**, and no locking around individual data changes – instead the `_emit()` method uses a `QMutexLocker` to safely check `_batch_mode` and queue signals [30]. This is exactly the "correct pattern" outlined in the plan [31]. We verified `_emit` is implemented as per plan: it locks briefly to append signals if in batch mode, and otherwise emits immediately outside the lock [32].
- The **"wrong pattern"** (explicit lock/unlock in setters) is indeed not present – we found no instances of `self._mutex.lock()` in `ApplicationState` methods.
- Batch operations in `ApplicationState` are implemented with `begin_batch()`/`end_batch()` and a context manager, which use the mutex correctly and emit all signals at once at the end [33] [34]. This aligns with the plan's design and ensures thread-safe signal emission when batching.
- **Thread-safety – StateManager:** As expected, `StateManager` has **no QMutex** and no thread assertions. It's intended for main-thread use only (UI layer), which is consistent with the plan. We did not find any misuse of threads in `StateManager`; it interacts with UI widgets and `ApplicationState` (which itself guards threads) for any data changes. The plan's Principle 3 is upheld: `ApplicationState` is thread-safe, `StateManager` is used only in the main thread.
- **Signal forwarding/delegation:** The current code already demonstrates some of the planned signal delegations:
- `StateManager` connects to `ApplicationState.frame_changed` and simply re-emits it as its own `frame_changed` signal for the UI [35]. This means UI components can listen to `state_manager.frame_changed` (wired to sliders, etc.) and still be responding to the single source of truth from `ApplicationState`.
- `StateManager.selection_changed` is emitted after filtering the `ApplicationState.selection_changed` signal [36] [37]. The adapter `_on_app_state_selection_changed` ensures that only selections for the active curve/timeline are forwarded [38] [39]. This matches the plan's intention to avoid duplicate selection signals and only emit relevant UI selection changes.
- No StateManager signal exists for curve data changes (as noted), so any UI refresh on curve data must come from `ApplicationState.curves_changed`. In practice, we found places in controllers where `ApplicationState.curves_changed` is used. For example, the plan suggests connecting `curves_changed` to a UI handler (`_on_curve_data_changed`); in the current code, the `StateSyncController` uses `app_state.curves_changed` to update the curve store and UI (as evidenced by bug fixes in Phase 5) [40] [41]. So the single-source signal is in effect.
- **New signals to add (undo/redo/tool):** These are confirmed missing, and the plan outlines their addition in **Phase 3**. Once added, the code will need to emit them appropriately:
- We identified that `StateManager.set_history_state(...)` changes `_can_undo/_can_redo` but currently just sets flags with no signal [42]. The plan calls for emitting `undo_state_changed`/`redo_state_changed` here. This is a clear gap to fill – tests will likely

be added as in the plan (e.g., `test_undo_state_changed_signal`) to ensure these signals fire when history state changes [43] [44]. ⚠

- Similarly, `StateManager.current_tool` changes are not broadcast (just logged) [26]. The plan's migration will introduce a `tool_state_changed` Signal that should be emitted in the setter. Currently, anything responding to tool changes (if any) must poll or be manually updated; adding this signal will improve responsiveness. ⚠

Verification of Specific Plan Items

- **Method naming (active_curve vs get_active_curve_name): Verified.** The plan's correction is reflected in code – `ApplicationState.active_curve` is a property. All usages in `StateManager` and elsewhere call the property (e.g., `active_curve` is used inside `ApplicationState.set_track_data` in the plan snippet [45], and indeed the code property exists). No incorrect method call is present.
- **Thread safety pattern understanding: Verified.** We reviewed `_assert_main_thread` and `_emit` implementations in `ApplicationState`. They match the described pattern [30]. No evidence of the "wrong" locking pattern was found. The use of `QMutexLocker` inside `_emit` and batching confirms the plan's guidance is already implemented in code.
- **Locations to add signals and variables: Identified.** We located where the plan suggests inserting new code:
- New signal definitions in `ApplicationState` – after existing signals (~line 141 in code) – e.g., `image_sequence_changed: Signal()` would be added. Indeed, in code the signals block ends at line 139 [46]; inserting there is straightforward.
- New private variables in `ApplicationState.__init__` for `_image_files`, `_image_directory`, `_total_frames` – likely after the existing state fields. The code's `__init__` has those fields absent, and we see around line 148-155 where other state is initialized [47]. The plan's suggested lines (~166) align with adding them right before or after the batch mutex setup. This is clear and there are no conflicts – these can be added as planned.
- New signals in `StateManager` – the class definition signals are at ~line 38 in code [22]. We confirm we can add `undo_state_changed`, `redo_state_changed`, `tool_state_changed` there. There is no existing signal by those names, so no conflict.
- `_emit_signal` helper in `StateManager`: The plan likely uses a pattern similar to ApplicationState's `_emit`. The current code already has a `_emit_signal` method in `StateManager` (used for batch updates and certain setters) [48] [49]. We should verify if this is used for new signals: e.g., `StateManager.playback_mode` setter uses `_emit_signal` to emit the `playback_state_changed` signal on change [50]. We can leverage `_emit_signal` for the new signals as well, to support batch mode. The plan's tests even mention verifying signals use the `_emit_signal` (batch-friendly) [51]. So the infrastructure is in place.
- **Signal connection updates:** The plan outlines replacing any `track_data_changed` connections with `curves_changed`. Since `track_data_changed` doesn't exist in code, there's effectively nothing to replace. We should instead ensure that wherever the UI or other components need to know about data changes, they listen to `ApplicationState.curves_changed`. The code already does this in some places (e.g., timeline tabs subscribe to the store which ultimately is fed by `ApplicationState.curves_changed`). No listeners for a non-existent signal were found, so this part of migration is effectively already satisfied.
- **Phase 0 checklist items:**
- Verified `active_curve` property (above).
- Verified thread safety pattern (above).
- Located where to add signals (identified in class definitions).

- Located where to add instance vars (identified in `ApplicationState.__init__`).
- Read `_emit()` implementation – confirmed it matches expectations [32].
  (All items in the plan's Phase 0 list are addressed.)

Discrepancies & Clarifications

- ⚠ **Duplicate Signal Reference:** The plan refers to `StateManager.track_data_changed` as a problematic duplicate. In the current code, this signal is already removed (no such attribute in `StateManager`). The concern about duplicate curve-data signals is valid, but any plan instructions to "remove `track_data_changed`" are effectively moot – it's already gone. We should focus on ensuring all components now rely on `ApplicationState.curves_changed`. No changes needed here except possibly updating documentation to reflect that `track_data_changed` is no longer in code.
- ⚠ `curves_changed` **Payload:** As noted, the mismatch in `curves_changed` signal payload (dict vs str) could affect migration steps. For example, the plan's example for connecting `curves_changed` to a slot that expects a curve name [52] would not work with the current signal (which sends a dict). In practice, the code's own controllers handle this by extracting what they need (e.g., `StateSyncController` iterates the dict of curves). We may need to **clarify if the plan expects to change the signal signature** or update the migration instructions to use the dict. Likely we keep the dict (multi-curve design) and adjust the plan's pseudocode accordingly. This is a point to clarify before implementing Phase 1. **?**
- ⚠ **Plan vs Code Signal Names:** The plan (in an appendix or code block) mentions signals like `selected_curves_changed` and `show_all_curves_changed` for selection state [53]. The current code instead has a combined `selection_state_changed: Signal(set, bool)` in `ApplicationState` [54] (to signal the set of selected curves and the show_all flag together). The unified signal is likely an updated approach. We should clarify if the plan wants to split those or continue using the combined signal as in code. The code seems to be working with the combined signal, so this may just be a documentation inconsistency. **?**
- ⚠ **File Reference (**`data_operations.py`**):** As mentioned, clarify that `services/data_service.py` is the actual file for data ops. Ensure migration tasks targeting file I/O update (like using `state.set_track_data` vs old methods) use the right references. This is a minor inconsistency in naming. **?**
- ⚠ `_original_data` **handling:** The plan explicitly defers migrating `_original_data` to a later phase [55], and we confirm it's still in `StateManager` (as `_original_data` list with a setter) [56]. This is **an intentional gap**. It means after this migration, `StateManager` will still hold `_original_data`. The plan notes this is blocked by a future multi-curve undo design. We just need to ensure we don't mistakenly move or alter `_original_data` now. Documenting this in code comments (as already done in the plan) is wise. (No discrepancy, just a noted omission.)
- **? UI Updates for New Signals:** Once we add `undo_state_changed` and `redo_state_changed`, the plan shows hooking them up to enable/disable the undo/redo toolbar buttons [57]. We should clarify if the main window already has those buttons and how they should be updated:
- We found `MainWindow.undo_button` and `redo_button` attributes (QPushButton) [58], but currently they might not be actively enabled/disabled on state changes. The migration will introduce that behavior. We need to ensure that after adding signals, we implement something like `undo_button.setEnabled(...)` on signal, as in the plan. No conflicting code exists now, so it should be straightforward. Just ensure to connect the signals in the `SignalConnectionManager` or `MainWindow` after creating the StateManager.

- **? Tool state usage:** The `tool_state_changed` signal will be new. We should clarify what in the UI should respond to it. Possibly the tool palette or indicators should update when the current tool changes. Currently, `current_tool` is used (for example) in the status bar or not at all – we didn't find explicit UI logic for tool switching aside from setting the value. The plan marks it "Important" (🟡) rather than critical, implying the app can function without the signal for now but it's good to have. We'll implement it, and perhaps no existing code will use it until we connect something (maybe future-proofing for when tool icons or menus reflect the state). No conflicts expected, just need confirmation on intended use.
- **? Testing assumptions:** The plan outlines several tests (for new signals, delegation behavior, etc.). We should verify if similar tests already exist or if they will be new. For instance, we saw no current test explicitly for `undo_state_changed` (since it's not implemented yet), so those will be new tests we add. We should be prepared that the plan's expectations (like `state_manager.set_track_data` ultimately triggers `app_state.curves_changed`) need to be met for tests to pass. Ensuring our implementation follows the plan's intended signal emissions is crucial. No code discrepancy here, but highlighting that our changes must make those tests pass.

Conclusion

Overall, the current codebase is largely **consistent with the migration plan's goals**. We verified that the problematic hybrid state is real – `StateManager` still contains application data (`track_data`, images, etc.) that should reside in `ApplicationState`. The thread-safety model in `ApplicationState` is correctly implemented (no changes needed there, just understanding). The major work will be relocating those properties and introducing the new signals:

- We have **confirmed** the existence and location of all elements mentioned in the plan: files, classes, properties, and methods. Key classes (`ApplicationState` and `StateManager`) and their attributes are in place as expected.
- We identified a few **outdated references** in the plan (e.g. nonexistent `track_data_changed` signal, a different `curves_changed` signature). These need minor plan adjustments but do not affect the core migration – just something to be mindful of when executing (we'll use the actual code's signals/types). ⚠
- There are **missing elements** that the plan intends to add (the 3 new signals, legacy setter/getter in ApplicationState for track_data, image sequence state in ApplicationState). Our audit found no obstacles to adding them – the code is ready to accept these changes.
- No unexpected conflicts were found between the plan and code; in fact, some migrations (selection state, removal of duplicate signals) have partially occurred already. This reduces risk.
- **Clarifications needed:** Primarily around how to handle the `curves_changed` signal data and ensuring tests expectations match the final implementation. It would be wise to double-check with the team whether to modify the signal to emit `curve_name` or keep it as `dict` and adjust test logic accordingly. **?**

With these points addressed, the migration can proceed with a clear understanding of what exists and what needs to change. Each phase of the plan has a clear mapping in the code, and we've highlighted any discrepancies to resolve before coding. The audit did not uncover any unknown surprises in the code – everything aligns with the architecture described. We can move forward confident that the plan is applicable, with only minor tweaks for current code realities.

---

1 5 14 15 19 20 28 29 30 32 33 34 46 47 54 application_state.py
https://github.com/semmlerino/CurveEditor/blob/fa40ae0b263017ea86e1ba9df8ce8b56a1c96668/stores/application_state.py

2 6 7 8 9 10 12 13 16 22 23 24 25 26 27 35 36 37 38 39 42 48 49 50 56
state_manager.py
https://github.com/semmlerino/CurveEditor/blob/fa40ae0b263017ea86e1ba9df8ce8b56a1c96668/ui/state_manager.py

3 4 data_service.py
https://github.com/semmlerino/CurveEditor/blob/fa40ae0b263017ea86e1ba9df8ce8b56a1c96668/services/data_service.py

11 17 18 21 31 43 44 45 51 52 53 55 57 STATEMANAGER_COMPLETE_MIGRATION_PLAN.md
file://file-EeueQ3Na9GgGCB4GbiZUr2

40 41 PHASE_5_VALIDATION_COMPLETE.md
https://github.com/semmlerino/CurveEditor/blob/fa40ae0b263017ea86e1ba9df8ce8b56a1c96668/PHASE_5_VALIDATION_COMPLETE.md

58 main_window.py
https://github.com/semmlerino/CurveEditor/blob/fa40ae0b263017ea86e1ba9df8ce8b56a1c96668/ui/main_window.py