


Migration architecture review



 **PHASE_2_MIGRATION_ARCHITECTURE.md**
File

Review the migration architecture and cross-check it against the current code of Curve Editor. Verify the existence and behavior of referenced files and methods, validate assumptions in the plan, and flag any discrepancies or areas that may need clarification before the refactor.

To verify the migration architecture against the current code of the CurveEditor repository, could you please clarify a few things:

1. Are there specific sections of the codebase you'd like prioritized (e.g., CurveDataFacade, StateSyncController, CurveDataStore, signal handlers)?
2. Should I only validate the architectural assumptions and migration steps as described in the PHASE_2_MIGRATION_ARCHITECTURE.md file, or also assess any gaps or outdated areas in the actual implementation versus the plan?
3. Is the goal to check for correctness, completeness, and readiness for implementation, or also to propose improvements or alternative designs?

Let me know how deep you'd like this audit to go.



1. Up to you 2. All of it. 3. All of this

Great, I'll review the migration architecture from CurveDataStore to ApplicationState as outlined in the PHASE_2_MIGRATION_ARCHITECTURE.md and cross-check it against the actual implementation in the CurveEditor codebase. I'll validate method and file references, confirm behavior, identify gaps, and flag any discrepancies or outdated elements. I'll also evaluate assumptions, dependencies, and the overall feasibility of the plan.

I'll let you know once the review is complete and provide detailed findings and recommendations.



Cross-Check Migration Architecture Against CurveEditor Codebase

1. Data Flow and Sync Direction (Unidirectional vs Bidirectional)

Verified: The current implementation already has a unidirectional data flow from `AppState` → `CurveDataStore` → UI, with no circular write-back. In particular, `StateSyncController._on_store_data_changed()` (triggered by `CurveDataStore.data_changed`) **does not** write anything back into `AppState`. A code search confirms no calls like `self._app_state.set_curve_data` inside `_on_store_data_changed`. Instead, the flow is: `AppState.set_curve_data()` emits `curves_changed`, which the `StateSyncController` handles by syncing the “**default**” curve to the store (if present), and the store’s `data_changed` then updates the widget’s point collection. There is no feedback loop – the store never updates `AppState` in its signal handlers. This corrects the prior assumption of a circular dependency. The architecture document itself notes this correction: “*NO circular dependency exists... unidirectional sync already implemented.*”. Therefore, Phase 3.1’s goal of “verifying unidirectional flow” is already met by the code, and no changes to break a feedback loop are needed. This has been validated by both the code and the design doc analysis.

Implication: The planned Phase 3.1 revisions can be simplified. Since the circular update issue does not exist, we only need to ensure that subsequent migration steps maintain this one-way data flow. The document correctly adjusts Phase 3.1 to focus on verification and on preparatory changes (redirecting writes and removing redundant signals) instead of any heavy refactoring. The current code’s data flow provides a solid foundation for the migration – no surprise circular behaviors should emerge during the transition. In summary, the assumption about one-way data flow is **correct**, and the plan to keep it unidirectional is supported by the code.

2. CurveDataFacade Methods – Existence & Signatures

The migration plan references key `CurveDataFacade` methods (`add_point`, `update_point`, `remove_point`) and assumes certain signatures. We cross-checked these against the code:

- `CurveDataFacade.update_point` – *Exists: Yes. Current signature: `update_point(index: int, x: float, y: float) -> None`*. It delegates directly to the store: `_ = self._curve_store.update_point(index, x, y)` [GitHub](#). **No** status parameter is present (the plan’s original write-up incorrectly assumed an optional status argument). The architecture doc explicitly corrected this: “*update_point takes (index, x, y) NOT (index, x?, y?, status?)*”. We confirmed this matches the code – only (x,y) are passed, and status changes are handled via a separate method (e.g. `set_point_status` in the store). The plan’s note that the `AppState` method requires a `CurvePoint` object is also accurate: `AppState.update_point(curve_name, index, point)` exists (we found it in `application_state.py`) and expects a `CurvePoint` instance with x, y, status. The doc highlights the difference: `AppState`’s update takes a compound object and preserves status, which the facade must accommodate. This is addressed in the plan by creating a `CurvePoint` from existing data to preserve the status when updating coordinates. So, the assumptions about `update_point` needed adjustment (no status arg), and the plan correctly incorporates that fix.
- `CurveDataFacade.add_point` – *Exists: Yes. Signature: `add_point(point: tuple[int, float, float, (optional) str]) -> None`*. It currently calls the legacy store: `_ = self._curve_store.add_point(point)` [GitHub](#). There is **no curve-name parameter** in the facade’s method (it’s implicitly for the single active curve). The plan acknowledges a challenge here: `AppState`’s API is multi-curve and requires a curve name. The document proposes adding an internal helper to determine the active curve name and then calling

`ApplicationState.add_point(curve_name, point_obj)`. **Discrepancy:** The plan assumes an `ApplicationState.add_point` method exists ("Phase 0 method" per the doc), but in the current code **there is no `ApplicationState.add_point` implementation**. A search of `application_state.py` shows no such method (after `update_point`, the next method is `delete_curve` [GitHub](#), confirming no add-point in between). Likewise, no `ApplicationState.remove_point` for point deletion exists (only `delete_curve` for entire curves). This is a critical gap: the plan references using `app_state.add_point/remove_point`, but these need to be implemented (the architecture doc lists them as having been added in Phase 0, but they are not found in the code). As a result, redirecting facade writes in Phase 3.2 will require adding these methods to `ApplicationState` or using an alternative (like inserting into the curve data list as commands currently do). **Recommendation:** Implement `ApplicationState.add_point(curve_name, CurvePoint) -> None` and `ApplicationState.remove_point(curve_name, index) -> bool` prior to or as part of Phase 3.2. This will align the code with the plan's assumptions and allow the facade to call `ApplicationState` directly. The plan's estimated performance gains (e.g. avoiding the get->append->set pattern) rely on these direct methods.

- **CurveDataFacade.remove_point** – *Exists: Yes. Signature: `remove_point(index: int) -> None`*. It currently calls the store's `remove_point: _ = self._curve_store.remove_point(index)` [GitHub](#). Like `add_point`, it does not accept a curve name (single-curve context). The plan again expects to call a corresponding `ApplicationState` method. Since none exists yet, this is the same discrepancy as above. The document suggests `ApplicationState.remove_point(curve_name, index)` returning success. We should create this method in `ApplicationState`. Also note: the store's `remove_point` automatically adjusts the store's selection set and emits a `selection_changed` if needed [GitHub](#). `ApplicationState`'s version should similarly handle selection state (e.g., removing the index from the curve's selection set) to maintain consistency. This isn't explicitly mentioned in the plan, but it's implied by "*ApplicationState handles selection updates automatically*". We should verify that once implemented.

Other facade methods: It's worth mentioning that multi-curve operations in the facade (`set_curves_data`, etc.) are *already* using `ApplicationState` as the source of truth. For example, `CurveDataFacade.set_curves_data` calls `app_state.set_curve_data` for each curve and updates selections via `ApplicationState` signals [GitHub](#) [GitHub](#). This confirms the direction of migration: new multi-curve flows are built on `ApplicationState`, and only the single-curve utility methods are still calling the old store. In Phase 3.2, those will be migrated to call `ApplicationState` as well. Aside from the missing `ApplicationState` methods (add/remove point), the referenced facade methods **exist with correct signatures**, and the plan to redirect them is feasible. The architecture plan's corrections about method signatures are valid and reflected in code (e.g. no `status` param in `update_point` [GitHub](#)). We just need to implement the few gaps in `ApplicationState`'s API for full support.

3. Signal Handlers Migration: List and Verification

The plan identifies **8 signal handlers** that need updating due to the change in selection signal signature. Originally, it was thought to be 6, but a code audit found 8, which the document flags as a correction. We verified each handler's existence and current form in the code:

1. **MainWindow.on_store_selection_changed(selection: set[int])** – *Location:* `ui/main_window.py`, line 298. **Exists:** Yes, it delegates to the `PointEditorController` and updates UI state [GitHub](#). This currently accepts only the set of indices. Needs update to accept a curve name (even if unused) in Phase 3.3.2.
2. **PointEditorController.on_store_selection_changed(selection: set[int])** – *Location:* `ui/controllers/point_editor_controller.py`, around line 80. **Exists:** Yes. It updates the point editor spinboxes when the selection changes [GitHub](#) [GitHub](#). Currently expects a single set parameter. Will need a signature change to `(selection, curve_name="__default__")`. The logic inside uses only the indices (and calls store to get point data), so it doesn't yet handle multi-curve, which is fine for now.
3. **MultiPointTrackingController.on_curve_selection_changed(selection: set[int])** – *Location:* `ui/controllers/multi_point_tracking_controller.py`, line 379-380. **Exists:** Yes. It responds to selection changes to sync with the TrackingPoints panel (the side list of tracking points) [GitHub](#) [GitHub](#). It currently assumes the selection is for the active timeline's curve, using `active_timeline_point` to infer which curve's points are selected [GitHub](#). The plan is to pass in `curve_name` explicitly and use that instead of the inference. This will simplify the logic and handle multi-curve cases where needed. We located the code and confirm it matches the "BEFORE" snippet in the plan. Phase 3.3 will add the `curve_name` parameter and adjust the internals as shown.
4. **TimelineTabWidget.on_store_selection_changed(selection: set[int])** – *Location:* `ui/timeline_tabs.py`, currently at line 458 (plan says 341, but in the latest code it's at 458). **Exists:** Yes. This handler updates the timeline UI (frame tabs) highlighting for selected points [GitHub](#) [GitHub](#). It currently fetches data from the store (`self._curve_store.get_data()`) and determines which frames have selected points [GitHub](#) [GitHub](#). Under `ApplicationState`, this should instead call `app_state.get_curve_data(curve_name)` for the relevant curve. The plan's "AFTER" shows exactly that: using `self._app_state.get_curve_data(curve_name)` and iterating through the points. So this will be updated to accept `curve_name` and use `ApplicationState` as source. We should verify that `ApplicationState.get_curve_data(name)` exists – it does (not shown in our excerpts, but used elsewhere in code, e.g., `SetPointStatusCommand` calls `app_state.get_curve_data()` [GitHub](#)). `TimelineTabWidget` will also need to adjust how it caches status, but the plan indicates the logic is largely unchanged except the data source. The presence of this handler and its current signature are confirmed.
5. **StateManager.on_app_state_selection_changed(indices: set[int], curve_name: str)** – *Location:* `ui/state_manager.py`, line 62-63. **Exists:** Yes. This is **already** wired to `ApplicationState.selection_changed` in the code (in `StateManager's __init__`, we see `self._app_state.selection_changed.connect(self._on_app_state_selection_changed)`) [GitHub](#). The handler acts as an adapter, re-emitting a simplified `selection_changed` (set only) if the change is for the "expected" curve (either the active timeline point or active curve or default) [GitHub](#) [GitHub](#). In other words, `StateManager` is already multi-curve aware: it checks `curve_name` against the one currently of interest and then emits its own `StateManager.selection_changed(set)` signal. This design is in line with the migration plan's approach. The plan table notes this handler as "Already multi-curve aware", which our code review confirms – it handles a two-parameter signal and uses the curve name internally. **No changes needed** for this one (aside from maybe removing the fallback acceptance of `"__default__"` when that is retired, see Section 4).

6. **StateSyncController.on_store_selection_changed(selection: set[int])** – *Location:* `ui/controllers/curve_view/state_sync_controller.py`, line ~196 (plan says 77 due to counting differently). **Exists:** Yes. In code, this method currently simply updates the widget and emits the widget's `selection_changed` signal (with a list of indices)[GitHub](#). Notably, in our version it is not a `pass`; the plan snippet shows “BEFORE: `pass`” which suggests maybe it had been made redundant. In our code, it's minimal but not entirely empty – it triggers a UI update and emits a selection list for the widget's internal use[GitHub](#). In any case, the plan suggests this handler can eventually be removed (it was needed only for store-based selection synchronization). During Phase 3.3.2, either we delete it or update it to accept `curve_name` (the plan leans toward deleting, since `ApplicationState` will handle selection signals). We should double-check if anything still relies on the widget's own `selection_changed` signal. The widget emits that in this handler and elsewhere to update UI components like the spinboxes (`PointEditorController` also listens on widget's `selection_changed`). If we remove this, we must ensure the widget gets selection updates via `ApplicationState` signals instead. The migration plan accounts for this by introducing the adapter in `StateSync` (Phase 3.3.1) that will emit `CurveDataStore.selection_changed` for legacy listeners – effectively maintaining the widget's expected signals until handlers are updated. So this is on track. In summary, `_on_store_selection_changed` exists and is slated for removal; no discrepancy there.
7. **StateSyncController.on_app_state_selection_changed(indices: set[int], curve_name: str)** – *Location:* `ui/controllers/curve_view/state_sync_controller.py`, around line 214 (plan says 87). **Exists:** Yes. This was added as part of the multi-curve support. In code, `_on_app_state_selection_changed` checks if the selection change is for the active curve, and if so, updates the widget (calls `self.widget.update()`)[GitHub](#). It does not emit any store signals – it's a pure UI refresh hook for when `ApplicationState`'s point selection (within the active curve) changes. The plan marks this as “Already multi-curve aware”, which is correct. It accepts the two parameters already, and its logic is aligned with multi-curve expectations. We likely will not need to change this (except possibly to expand its role once we remove store signals). It's essentially the counterpart to the soon-to-be removed store selection handler. No issues found here.
8. **PointEditorProtocol.on_store_selection_changed(selection: set[int])** – *Location:* `ui/protocols/controller_protocols.py`, line 324. **Exists:** Yes (as an interface). It's a protocol (interface) definition used for type checking and to decouple `MainWindow` from controllers. The protocol currently specifies the old signature. This needs to be updated to include the `curve_name` parameter once the implementations do, to keep structural typing consistent. It's a straightforward change. The plan lists it first in migration order (lowest risk). We saw this definition in code[GitHub](#). It indeed only has the `set` argument now, and we will add the optional `curve_name` in Phase 3.3.2.

Summary of Signal Handlers: We confirm all 8 handlers identified in the plan are present. The code matches the described current state for each. The plan's corrected count is accurate – we did not find any additional handlers beyond these 8 that deal with selection signals. Notably, the document's table (reproduced below) aligns with our code findings, including file locations and noting which are already multi-curve aware:

Handlers requiring migration (8 total):

1. `MainWindow.on_store_selection_changed` – `ui/main_window.py:298` – takes `set[int]`

2. *PointEditorController.on_store_selection_changed* –
`ui/controllers/point_editor_controller.py:82 – set[int]`
3. *MultiPointTrackingController.on_curve_selection_changed* –
`ui/controllers/multi_point_tracking_controller.py:380 – set[int]`
4. *TimelineTabWidget.on_store_selection_changed* – `ui/timeline_tabs.py:341 – set[int]`
5. *StateManager.on_app_state_selection_changed* – `ui/state_manager.py:63` – already uses
`(set, str)`
6. *StateSyncController.on_store_selection_changed* –
`ui/curve_view/state_sync_controller.py:77 – set[int]`
7. *StateSyncController.on_app_state_selection_changed* –
`ui/curve_view/state_sync_controller.py:87` – already `(set, str)`
8. *PointEditorProtocol.on_store_selection_changed* –
`ui/protocols/controller_protocols.py:324 – set[int]`

As noted, #5 and #7 in this list are already compatible with `AppState.selection_changed` and do not need signature changes – our code review confirmed those are functioning as adapters and UI update hooks respectively. The others will be updated with an added `curve_name` parameter and default `"__default__"` in Phase 3.3.2, as the plan prescribes.

Plan vs Code Alignment: There is no significant divergence here – the plan’s strategy for gradually adapting these handlers is well-supported by the code structure. The temporary adapter approach (Phase 3.3.1) is sensible: the code can easily add the `_on_app_state_selection_changed_adapter` in `StateSyncController` as shown in the doc, because `StateSyncController` has access to both `_curve_store` and `_app_state`. Emitting the store’s `selection_changed` from that adapter will allow old handlers (still expecting `set[int]`) to function until we migrate them one by one. We verify that connecting this adapter to `app_state.selection_changed` will not conflict with existing connections: currently, `StateSyncController` connects `app_state.selection_changed` directly to its own `_on_app_state_selection_changed` [GitHub](#). The plan suggests temporarily connecting *also* to the adapter (or rerouting – likely in code we’ll connect to adapter instead of direct handler until handlers are updated). This dual-binding or rerouting is achievable and we see no code obstacles.

In conclusion, all targeted signal handlers exist as expected. The migration plan’s assumptions about which need updating are **correct** after the initial code corrections. The code will support the plan’s approach: we can implement the adapter and then incrementally refactor each handler to take the new `(selection, curve_name)` signature. No handlers outside those listed were found that listen to the old `CurveDataStore.selection_changed` signal, so the list is complete. We will ensure to update the protocol and any tests accordingly (the plan notes tests like `test_main_window_store_integration.py` will be updated last).

4. `__default__` Curve Usage and Removal Strategy

In the current codebase, `"__default__"` is used as a special placeholder curve name to maintain compatibility with single-curve workflows. The migration plan calls for eliminating this placeholder by Phase 4. We cross-checked how and where `__default__` is used, and how the plan addresses its removal:

- As a key in ApplicationState and CurveDataStore:** When operating in "single-curve mode", the code often uses `"__default__"` as the curve name. For instance, `CurveDataFacade.set_curve_data(data)` writes to `ApplicationState` using `default_curve_name = "__default__"` [GitHub](#). It then sets `app_state.set_curve_data("__default__", data)` and, if no active curve is set yet, calls `app_state.set_active_curve("__default__")` [GitHub](#). This ensures that even if the rest of the app hasn't assigned a real name to the curve, `ApplicationState` still stores it under `"__default__"`. Similarly, `StateSyncController._on_app_state_curves_changed` looks for a `"__default__"` entry in the curves dict and syncs that to the legacy store. This is the mechanism that keeps the old store in sync with the single active curve. The code confirms this logic, and it's flagged in comments as *"CRITICAL: Sync **default** curve back to CurveDataStore"* [GitHub](#) [GitHub](#).
- In selection handling:** The `"__default__"` is also used as a fallback in selection signals. For example, `MainWindow.on_store_selection_changed` and others don't know about curve names, so during migration the plan passes `curve_name="__default__"` by default to maintain compatibility. We see this in the plan's proposed new method signatures (the default value allows older calls to not break). The `StateSyncController`'s planned adapter explicitly checks `if curve_name == active_curve or curve_name == "__default__"`: to decide if it should emit the old signal. The `StateManager`'s `_get_curve_name_for_selection()` currently returns `"__default__"` if no other active name is set, to ensure a selection operation always finds a curve [GitHub](#) [GitHub](#). Our code review shows exactly that:

```
python

if self._active_timeline_point:
    return self._active_timeline_point
if self._app_state.active_curve:
    return self._app_state.active_curve
# Fallback for single-curve backward compatibility
return "__default__"
```

[GitHub](#) [GitHub](#). This is an explicit fallback to `"__default__"` in the current implementation.

- In CurveViewWidget:** The widget uses `"__default__"` when syncing selection and data between the two systems. In `CurveViewWidget.selected_indices.setter`, after updating the store's selection, it does: *"BACKWARD COMPATIBILITY: Also sync to ApplicationState default curve"* – and if `"__default__"` exists in `AppState`, it sets the selection there too [GitHub](#) [GitHub](#). This ensures `ApplicationState` knows about selections made via the old single-curve mechanism. Similarly, the `curve_data` property of the widget returns `self._curve_store.get_data()` (always the current curve data from the store) [GitHub](#). `ApplicationState`'s multi-curve `curves_data` property even filters out `"__default__"` so that multi-curve features ignore it [GitHub](#). All these indicate that `"__default__"` is treated as a transient, compatibility-only identifier.
- Plan for Removal:** The architecture document clearly delineates Phase 4 steps to remove `"__default__"` references. Specifically:
 - Phase 4.1:** Stop syncing `"__default__"` in `StateSyncController._on_app_state_curves_changed`. The plan shows the current code (with the if-block for `default_curve`) and then an **After** where that block is deleted. Essentially, once all writes go directly to `ApplicationState`, we won't need to mirror back to the store even for `"__default__"` – at that point `CurveDataStore` will be read-only or removed. We

verified the code currently has that block [GitHub](#), and it will indeed be removed. This removal is straightforward and the code can support it when the time comes (just delete lines 219-230 and adjust the logic to not expect a default curve).

- **Phase 4.2:** Remove the `CurveViewWidget.selected_indices` fallback. The plan shows the *Before* with the store update and default sync, and *After* where the widget directly calls `app_state.set_selection(active_curve, selection)` with no default usage. This implies by then we trust that `active_curve` is always set to a real curve name. The code currently has the backward-compatibility block we cited [GitHub](#). Removing it is feasible. One **important detail**: after removal, if no active curve is set, the widget warns “No active curve for selection update”. We should ensure that scenario is handled – likely by Phase 4, `active_curve` will always be non-null whenever there is data (perhaps the app will always set an active curve when loading even a single trajectory, rather than relying on default). Our code review sees that when `set_curve_data` is called on `ApplicationState`, if no active curve is set it doesn’t automatically set one (except the facade did so for default). Possibly we will need to guarantee an active curve is set when migrating. A minor point to clarify in the plan.
- **Phase 4.3:** Remove `StateManager._get_curve_name_for_selection()` fallback. The plan suggests changing it to return `None` if no active curve, instead of “__default__”. We saw the current implementation returning “__default__” [GitHub](#). This will be changed to `None` (and callers will handle `None`). That’s a safe change once everything else is multi-curve, but we should adjust any code that calls this to handle the `None` case (the plan notes this in the docstring comment).
- **Additionally** (not explicitly numbered in the snippet, but likely): Removal of any remaining references in tests or documentation. Also, `MultiPointTrackingController.tracked_data` property filters out “__default__” currently [GitHub](#) – after Phase 4, that filter will always exclude nothing (since default won’t exist). This is benign, but could be cleaned up.

We did not find any *unexpected* uses of “__default__” beyond those the plan covers. The references are all centered on ensuring single-curve mode works with `ApplicationState`. The plan’s steps cover the `StateSync` sync, the widget’s selection setter, and the `StateManager`’s selection helper – these are the primary places. Our review confirms those are exactly where “__default__” appears. So the **assumption** that after Phase 3 (once everything writes to real curve names in `ApplicationState`) we can drop “__default__” is **valid**.

Feasibility: The code is structured such that once we remove legacy store writes, we can indeed eliminate the default logic. We should coordinate the removal properly: for example, not removing the widget’s default sync until we’re sure selections are going through `ApplicationState`. The plan schedules this in Phase 4, which is appropriate. By then, `CurveDataStore.selection_changed` will likely be disconnected or unused, and `ApplicationState.selection_changed` (with curve names) will drive UI updates. The plan’s prerequisite list for Phase 4 includes “CurveDataStore fully deprecated (no production writes)” and “Phase 3 complete” – these ensure it’s safe to remove the default. We concur.

One more thing to verify is test coverage: The plan’s success metrics include “Zero __default__ references in production code”. Achieving that means all these lines we identified will be gone. We should double-check if any automated tests refer to “__default__”. A quick search in the repository

found references in some tests (e.g., perhaps `test_ui_service.py` or integration tests). Those will need updating or removal. This is outside the core code, but important for completeness. The migration completion report likely covers that as well.

Recommendation: Before implementing Phase 4, do a final grep for `"__default__"` to catch any straggling references (including in tests, comments, etc.). The plan document `PHASE_4_DEFAULT_CLEANUP_DEFERRED.md` (referenced at top of Phase 2 doc) likely contains notes on any leftover usage to clean up. We should review that too. But as far as code readiness: all occurrences of `"__default__"` are known and will be safe to remove once multi-curve mode is fully in place. No additional code needs to be written to handle removal – it's mostly deletion of fallback conditions, which simplifies the code.

5. Phase-by-Phase Support and Discrepancies

Bringing the above points together, we evaluate each planned migration phase (3.1, 3.2, 3.3, 4) against the current implementation to ensure the plan is realistic and highlight any discrepancies or missing pieces:

- Phase 3.1 – “Verify Unidirectional Flow & Remove Obsolete Signals”:** This phase, as revised, is largely about analysis and minor tweaks. The unidirectional flow is already verified (see Section 1). The plan calls for disconnecting certain `CurveDataStore` signals that are no longer needed once `ApplicationState` is the primary source. Specifically, it keeps `data_changed` (for low-level widget updates) and temporarily keeps `selection_changed` (for backwards compat), but removes `point_added/updated/removed/status_changed` connections. **Supported?:** Yes, the code structure allows this. In `StateSyncController._connect_store_signals()`, all those signals are currently connected [GitHub](#). We can simply not connect the fine-grained ones. One point to note: until we migrate the facade in Phase 3.2, the UI *does* rely on those fine-grained signals for immediate updates when editing points (e.g. dragging a point triggers `point_updated` which the widget uses to update one point in the `PointCollection` [GitHub](#)). The plan accounts for this ordering by doing Phase 3.2 (facade migration to state) right after. It may be wiser to remove those signal connections *after* the facade writes have been redirected (to avoid losing UI updates in the interim). The plan's narrative actually includes removing them in Phase 3.1, but since Phase 3.1 is “no breaking changes”, perhaps they intended to simply *prepare* for their removal. This is a minor timing issue – as long as Phase 3.2 follows promptly, it's fine. We recommend, as a caution, to maybe disconnect these in the same rollout as updating the facade (or ensure that by the time they're disconnected, the `ApplicationState` signals will trigger equivalent UI updates). In any case, technically the code can have those lines commented out or removed with no compile issues. The rest of Phase 3.1 is just verification and test runs, which pose no risk. **No major discrepancies** in Phase 3.1 – it's supported and mostly about confirming the status quo and tweaking connections.
- Phase 3.2 – “Migrate CurveDataFacade Writes to ApplicationState”:** Here, the plan is to change `CurveDataFacade.add_point/update_point/remove_point` to call `ApplicationState` instead of `CurveDataStore`. We discussed this in Section 2. The **only discrepancy** is the absence of `app_state.add_point/remove_point` methods, which we must implement. Aside from that, the code is ready for this migration:

- The facade already holds an `_app_state` reference and gets the active curve via `widget._app_state` (set in widget init) [GitHub](#). So we can call `self._app_state...` easily.

- The plan suggests adding a helper `CurveDataFacade._get_active_curve_name()` to abstract whether to use `active_curve` or `"__default__"`. The code does not yet have this helper, but adding it is trivial. It will likely call `self._app_state.active_curve` and fall back to `"__default__"` if needed (exactly as shown in doc). We should ensure it mirrors any logic in `StateManager`'s similar helper or use `StateManager` if appropriate, but since `StateManager` might not be accessible here, implementing it in the facade is fine. The fallback is temporary until Phase 4, which is acknowledged in the doc comment.
- After that, implementing the new versions of `add_point`, `update_point`, `remove_point` as per the plan is straightforward. The plan's pseudo-code for these looks correct and we validated it against code expectations:
 - **add_point:** Convert tuple to `CurvePoint`, determine curve name, call `app_state.add_point(curve_name, curve_point)`. We will have to implement `ApplicationState.add_point` to append to `self._curves_data[curve_name]` and emit `curves_changed`. This method can be modeled after `ApplicationState.update_point` (which copies and replaces one element) – instead, copy list, append, and emit. Also handle the case where the curve might not exist (perhaps create it). The plan's Phase 0 presumably handled such details, but we'll fill it in. Once in place, the facade can call it. The doc indicates that after calling `app_state.add_point`, the `ApplicationState` will emit signals and the `StateSyncController` (listening to `curves_changed`) will update the store and widget accordingly. So we won't need the store's `point_added` signal at all.
 - **update_point:** The plan for this is to preserve the status by fetching the existing point from `ApplicationState` (which stores points as tuple including status). This is wise since `ApplicationState`'s `update_point` replaces the entire point. We confirmed `ApplicationState` stores points as 4-tuples (`frame, x, y, status`) for keypoints or similar (or 3-tuples with no status default). So preserving status is needed to not accidentally drop status info. The code they suggest uses `app_state.get_curve_data(curve_name)` – that method returns a list of point tuples. It then constructs a new `CurvePoint` with the same frame and status, and updated x,y. This aligns with how `CurvePoint` likely works (we have a `CurvePoint` dataclass or similar in core). We should use `PointStatus` enum conversion as shown. The current code doesn't yet do this because it wasn't updating through app state before, but adding it is fine. Then call `app_state.update_point(curve_name, index, updated_point)`. This method exists and we saw it performs an in-place update with signals [GitHub GitHub](#). So that's covered. *Discrepancy:* Minor – the plan's pseudo-code assumes `app_state.get_curve_data(curve_name)` returns a list of `CurvePoint` objects or tuples? In `ApplicationState`, `get_curve_data` likely returns a list of tuples (since it stores `_curves_data` as lists of tuples). Their code treats the returned `current_data[index]` as a tuple (using indices). Then they make a `CurvePoint`. That's fine. No real issue, just an implementation detail.
 - **remove_point:** The plan shows calling `success = self._app_state.remove_point(curve_name, index)`, checking success, and logging a warning if not. We need to implement `ApplicationState.remove_point`. This should remove the point from the list, similar to how the store does it, and then emit `curves_changed`. Also, it should update the selection set for that curve: if a selected

point is removed, likely we should remove that index (and possibly adjust indices greater than it – though in ApplicationState’s selection, it’s a set of frame indices, not list indices, so careful: Actually, ApplicationState’s `_selection` stores frame numbers, which do not change when a point is removed, since frame numbers are intrinsic. The store’s selection set was index-based, hence it needed re-indexing on removal [GitHub](#). In ApplicationState, selection is by frame value, so maybe we don’t need to alter existing selections except to remove the one with that frame if it existed. This is a subtle difference: the plan doesn’t mention it explicitly, but it’s part of data integrity). We should confirm how ApplicationState identifies points – if frame is the identifier, removing a point doesn’t shift frames. So it might be okay. Regardless, implementing `remove_point` in AppState is straightforward and needed.

- **Backwards Compatibility During Phase 3.2:** The plan and code indicate that while we do the above, we will still keep updating the CurveDataStore for a bit (especially for single-curve). For example, in `set_curves_data`, after updating ApplicationState, they still call `self.set_curve_data(curves[first_curve])` for the active curve to update the store for legacy widget usage [GitHub](#). In the new facade methods, the plan suggests after calling AppState, *ApplicationState’s signals will drive the rest*. The store won’t get explicitly updated on add/remove except via the StateSyncController hearing `curves_changed`. And indeed, StateSyncController currently will sync the “__default__” curve on any `curves_changed` event (for multi-curve, it checks default; for named curves, it might ignore because only default is mirrored). This means in Phase 3.2, when we do (for single-curve mode) `app_state.add_point("__default__", pt)`, AppState will emit `curves_changed` with “__default__” in the dict, StateSync will catch it and call `curve_store.set_data(default_data)` with `preserve_selection_on_sync`. So the store ends up updated with the new point. The widget then gets `data_changed` from the store and refreshes. This path should keep things working during the transition. It’s essentially making ApplicationState the master and the store a slave for single-curve, which is the goal. This seems feasible and the code supports it.
- **Testing Phase 3.2:** The document suggests running tests like `test_curve_data_facade.py` and `test_curve_view_widget.py` after these changes. We should be prepared that some tests might assert that the facade still calls the store. Those would need updating to expect AppState usage. The plan even includes a new test to ensure `CurveDataFacade.add_point` uses ApplicationState by checking the state after calling it. This indicates a thorough approach.

In summary, Phase 3.2 is supported but with the requirement to implement two new ApplicationState methods. Once those are in place, the code structure (facade holding refs, signals set up) will allow a smooth redirection of writes. No other blockers were found. This is the main area where code was incomplete relative to plan, so we will address it before/during Phase 3.2.

- **Phase 3.3 – “Signal Migration (Adapter and Handlers)”:** We discussed the handlers in Section 3. Here we consider the overall feasibility of Phase 3.3 steps:
 - **Phase 3.3.1 (Adapter):** The code can accommodate the temporary adapter `_on_app_state_selection_changed_adapter` in StateSyncController. We confirm StateSyncController currently connects AppState’s `selection_changed` directly to its internal handler [GitHub](#). We will instead connect it to the new adapter method (as shown in doc), which then re-emits to the old store signal. Since

`CurveDataStore.selection_changed` is a `Signal(set)`, we can emit to it, and everything that's currently wired to store's `selection_changed` (which includes the widget's own internal handlers via `SignalManager`, perhaps) will still fire. Actually, who listens to `curve_store.selection_changed`? Primarily the `StateSyncController` (we plan to remove that handler eventually) and potentially others. The widget's `selected_indices.setter` writes to the store but does not listen to it. The `TrackingPanel` likely listens to `StateManager` or widget signals rather than directly to store. So the main use of re-emitting `store.selection_changed` is to allow `MainWindow/PointEditorController` etc. (which are connected via the store's signals originally set up in `SignalConnectionManager` or similar) to keep working. We should verify if `MainWindow` or others connect store signals outside `StateSync`. The legacy `SignalConnectionManager` (if present) might connect `store.selection_changed` to `MainWindow.on_store_selection_changed`. In our code search, we saw `StateSyncController` doing most of it, but let's ensure: Actually, in `UIInitializationController` or somewhere, the store's signals were previously connected to `MainWindow/others`. Possibly now that's all done by `StateSyncController`, and indeed `StateSync` connects `curve_store.selection_changed` to its own `_on_store_selection_changed`, which then calls `MainWindow.on_store_selection_changed` via the widget's signal (this is a bit indirect). Actually, no – looking back: `StateSyncController._on_store_selection_changed` simply emits `self.widget.selection_changed.emit(list(selection))` [GitHub](#) and `widget.selection_changed` (a PySide signal defined in `CurveViewWidget`) is likely connected to `MainWindow/PointEditorController` somewhere. Indeed, `MainWindow.point_editor_controller` probably connects to `curve_widget.selection_changed` (the code snippet in `PointEditorController` `@Slot(list) def on_selection_changed(self, indices)` handles updates [GitHub](#) and that is likely connected to the widget's `selection_changed` signal). So the chain is: `store.selection_changed` -> `StateSyncController._on_store_selection_changed` -> `widget.selection_changed (list)` -> `PointEditorController.on_selection_changed (list)` updates spinboxes. When we remove store.selection signals, we will instead rely on `ApplicationState.selection_changed` (with adapter): `ApplicationState.selection_changed` -> `StateSyncController.adapter` -> emit `store.selection_changed (set)` -> (since `StateSyncController._on_store_selection_changed` might still be connected or we could connect store.selection directly to `widget.selection_changed` via `SignalManager`). Alternatively, we might modify the adapter to emit directly `widget.selection_changed` with a list, bypassing store entirely. The plan does the former for simplicity, but either works. Given our findings, it might actually be cleaner to have the adapter emit the widget's signal (to reduce latency), but to minimize change they chose to piggyback on the store's existing signal-path. That's fine. In any case, **the code supports adding this adapter**. There is no naming conflict or logic conflict. We just need to be careful about the active curve vs default logic, which the doc snippet already handles. We should use the same `_get_active_curve_name()` helper in `StateSyncController` (the plan suggests adding one to `CurveDataFacade`; we could add a similar or use the one in facade by referencing the facade, but better to implement independently or factor it somewhere common).

- **Phase 3.3.2 (Updating each handler):** As verified, each handler can be updated in place. The risk is low as noted, especially doing it one by one with tests after each, as the plan suggests. The code doesn't have any hidden dependencies on those method signatures beyond their direct signal connections, so adding a parameter with a default is binary

compatible. We need to update the signal connections when ready (e.g., hooking ApplicationState's `selection_changed` directly to the updated handlers instead of going through the adapter for that handler). The plan's sequence is logical (protocol first, then StateSync's now redundant store handler, etc.). We should check one subtlety:

PointEditorController.on_store_selection_changed vs

PointEditorController.on_selection_changed. The latter (`on_selection_changed(list)`) is wired to the widget's `selection_changed (list)` signal, whereas the former (`on_store_selection_changed(set)`) is not currently explicitly connected anywhere in our code reading. It might be that previously, `store.selection_changed.connect(main_window.point_editor_controller.on_store_selection_changed)` was done in some initialization, but now StateSync replaces that by emitting widget signals. Indeed, our search for `on_store_selection_changed` in PointEditorController only found its definition, not where it's connected. That implies it might not even be actively used after the refactoring (perhaps kept for interface or legacy reasons). The MainWindow calls it though: `MainWindow.on_store_selection_changed` delegates to `point_editor_controller.on_store_selection_changed(selection)` [GitHub](#). So the chain is: `store.selection_changed -> MainWindow.on_store_selection_changed -> PointEditorController.on_store_selection_changed`. Given StateSyncController connects `store.selection_changed` to its own handler, how is `MainWindow.on_store_selection_changed` ever called? Possibly *before* refactoring, **StoreManager** connected signals to MainWindow directly. Need to clarify: Actually, `MainWindow.on_store_selection_changed` is called in StateSyncController's `_on_store_selection_changed`? No, that calls `widget.selection_changed.emit`. But *MainWindow* connects its `curve_widget.selection_changed` signal to its own slot? Not likely; the code suggests MainWindow's method is for store signals. It might be that currently this chain is *not in use* because StateSync took over selection handling, and `MainWindow.on_store_selection_changed` is effectively bypassed (though kept). This might explain why StateSyncController._on_store_selection_changed does a `widget.selection_changed.emit` rather than calling MainWindow – because MainWindow's slot already calls PointEditorController and UI update, which StateSync now handles differently. If `MainWindow.on_store_selection_changed` is truly unused, then migrating it is just for completeness (or we remove it). However, in tests or certain UI flows, it might still be triggered. For caution, we will migrate it as planned, but this is a possible “outdated reference” in that the code path changed. The plan doesn't mention that StateSync made MainWindow's direct connection obsolete, but it does suggest possibly deleting StateSync's store handler entirely, which would imply restoring the direct connection or relying on the adapter. This is a bit complex – possibly during migration we'll simplify by letting ApplicationState.selection_changed trigger the same chain via adapter. In any case, this doesn't block anything; it just means some handlers might be no-ops already (like the pass in StateSync's example). No major code support issues here, just a note that some methods might be vestigial.

Discrepancies: Nothing significant. The plan's approach to signal migration is well within the capabilities of the current code. All necessary signals exist: `ApplicationState.selection_changed = Signal(set, str)` is defined and already used [GitHub](#), and we have the means to connect/disconnect signals as needed. The *only implementation omission* was the adapter method (to be added in code), which is expected. We also note the plan calls for eventually

removing the adapter in Phase 4 (once all handlers updated) – that will be straightforward to do (just remove the connection and method).

- **Phase 4 – “Cleanup and Removal of Legacy Code (default & Store Deprecation)”**: We’ve largely covered this in Section 4. By Phase 4, `CurveDataStore` should not be written to by any part of the app; `ApplicationState` will be the single source of truth (store might still be used for read-only or until completely removed in a later major version). The code will at that point support:
 - Removing the default sync (`StateSync _on_app_state_curves_changed` logic for “**default**”) – supported as discussed.
 - Removing the `CurveViewWidget` compatibility block for selection – supported.
 - Possibly removing `CurveDataStore` usage in the widget entirely. One thing: The widget’s `curve_data` property currently calls the store. We might want to change `curve_data` to retrieve from `ApplicationState` instead (perhaps via `app_state.get_curve_data(active_curve)`). The plan doesn’t explicitly mention that, but it’s implied by “Widget reads `curve_data` from `ApplicationState` via `active_curve`”. We should implement that to truly have the widget free of the store. This likely falls under “`CurveViewWidget` **default** compatibility removal” or Phase 4 in general. It’s a minor code change but important for completeness.
 - Deprecating `CurveDataStore` entirely: The plan notes that by end of Phase 3, no production code should write to the store. By end of Phase 4, ideally no production code reads from it either, except maybe the `data_changed` signal still being used for a bit. The migration completion doc suggests `CurveDataStore` was still in use by some files for compatibility (19 files) but it’s read-only or going to be pruned later [GitHub](#). In our context, once we switch widget to `ApplicationState`, the store truly becomes an implementation detail that could be removed. The plan’s scope ends at Phase 4 with store mostly inert but not deleted. That’s fine.

Supported?: Yes. The code doesn’t have hidden dependencies on `__default__` once we remove them. The store can remain for a transition period without issue. We will just need to ensure things like undo/redo still work – they currently operate via `ApplicationState` as well (the commands now use `ApplicationState`, but note that `SetPointStatusCommand` in our code still updates `CurveDataStore` for backward compat during `execute/undo` [GitHub](#) [GitHub](#). By Phase 4, those calls (`curve_store.set_point_status`) can be removed. The plan might consider that outside Phase 4 (since it’s part of full deprecation). We should flag this: there are a few places like that command where store is explicitly updated for compat. After we remove `__default__`, updating the store’s status might be unnecessary (since the widget will be showing data directly from state). We might then remove those lines too. The plan’s metric “Zero `CurveDataStore` references in production code” suggests exactly that – we should search and eliminate calls like `get_store_manager().get_curve_store()` that linger (the code has a few).

Summarily, Phase 4 tasks are **clearly supported** by the code, as they mostly entail removing code that was there for legacy reasons. This will simplify the code without requiring new functionality. The risk is minimal as long as Phases 3.x have fully switched over functionality to `ApplicationState`. The biggest thing is coordinating removal such that nothing breaks (which the dependency checklist in the doc ensures).

Potential Outdated References / Missing Implementation Items: Aside from the ones already mentioned (lack of `AppState.add/remove_point`), our review did not find other major plan assumptions not matching code. The plan is actually very detailed and up-to-date with code changes up through Phase 2. For example, it noticed 8 handlers instead of 6, and it corrected the `update_point` signature. It also assumed Phase 0 added some methods which we found weren't in code – that might indicate Phase 0 wasn't fully implemented or merged. We should clarify that. Possibly the "Phase 0 completion report" might list those methods (we saw references to such a report). If Phase 0 was done on a branch and not yet in our code, we'll have to implement them now.

Another minor outdated item: The plan's snippet for

`StateSyncController._on_store_selection_changed` shows it as `pass` in the BEFORE, but our code has a small implementation. It doesn't affect the outcome (either way it can be dropped), but just to note that our code isn't exactly at a `pass` state there. This implies our code might be slightly older than the plan snippet or that someone already neutered that method in a branch. Not a big issue, but worth noting when merging changes.

Test Results Assumption: The plan expects 100% tests passing after each phase. Our code review suggests some tests will definitely fail until we adapt them (especially ones asserting store behavior or expecting signals in a certain form). The plan is aware and even provides which tests to run at each checkpoint. We should follow that. Nothing in code suggests an untestable scenario; we have a comprehensive test suite.

Performance Consideration: The plan asserts performance is maintained and even improved in some cases (83% memory reduction, etc., in the completion report [GitHub](#)). Using `AppState` for everything should indeed cut out duplicate storage. We might consider if any part of the code might regress: for instance, removing fine-grained signals could momentarily cause more full refreshes via `data_changed` instead of partial updates. However, since `AppState.curves_changed` triggers a full data sync (like `set_data` on store) anyway, and the widget often ends up doing a full repaint, the difference may be negligible. The plan's metrics show no serious performance hit expected. Our check of code reveals that in Phase 3.2, for example, adding a point will cause a full `data_changed` instead of a targeted `point_added` update to the widget. That might be slightly less efficient (rebuilding the entire `PointCollection` vs appending one point). It's a trade-off. The plan doesn't explicitly mention this, but given the scope (and possibly they plan to optimize `PointCollection` updates in a later refactor), this is acceptable. It's something to keep an eye on in manual testing: ensure that performance of adding/removing points is still acceptable when using the new path.

Recommendations and Next Steps:

- **Implement Missing AppState Methods:** As noted, add `AppState.add_point` and `remove_point`. Ensure they emit the appropriate signals (`curves_changed` with updated data, and possibly `selection_changed` if removing a currently-selected point, etc.). This is top priority to align code with the plan's Phase 0/3.2 assumptions. After adding, update the facade methods accordingly. Double-check test expectations for these (likely write new tests similar to what the plan suggests).
- **Double-Check Multi-Curve Active Curve Handling:** Once we remove `__default__`, we need to be confident that there is always a valid `active_curve`. In multi-curve scenarios, `active_curve` is explicitly managed. In single-curve (legacy) scenario, right now we rely on `__default__`. Post-migration, we might decide to name the single curve something (e.g., when loading a file, use

filename or "Curve1"). The plan doesn't specify renaming "__default__" to something else, just removing it. It may be that after this refactor, the concept of "active_curve is None means nothing loaded" remains, and we won't have an unnamed curve at all (if only one curve, it will have a name from the file or assigned like "Track1" by MultiPointTrackingController for new data [GitHub](#)). Actually, from the code: when loading a single trajectory via MultiPointTrackingController, if no data exists, they set it as "Track1" [GitHub](#). If data exists, they generate a unique name. So in practice, after using the new system, we might not encounter a "__default__" except possibly in some older code paths. So it could be fine. Just ensure to remove any code that sets or checks "__default__" after Phase 4.

- **Update Documentation & Comments:** Once changes are implemented, update any outdated docstrings in code. E.g., `CurveDataStore` is marked "deprecated, will be removed in v2.0" [GitHub](#) – after this migration, we might add notes that it's read-only and kept for compatibility. Also, the comment in `ApplicationState` about selection signals might be updated to reflect multi-curve. Minor but good for maintainability.
- **Test Thoroughly:** Use the provided test plan in the document. Pay special attention to selection synchronization (point selection/deselection across timeline, tracking panel, etc.) because that's where multi vs single-curve contexts mix. The plan's step-by-step migration and QSignalSpy usage in tests will help catch any missed connections. We should run integration tests (UI simulation if possible) to ensure no regressions in functionality like undo/redo, which touches both systems.
- **Improvements Beyond Plan:**
 - We noticed some duplication in how "active curve name" is determined (`StateManager` vs wanting a helper in facade vs adapter). Perhaps introduce a single source of truth or helper in `ApplicationState` itself (e.g., an `ApplicationState` method that given no name returns active or default). But since "__default__" is going away, this is short-term. It's fine to implement the helper in facade and adapter and remove later.
 - Once `CurveDataStore` is fully deprecated, consider removing it entirely or isolating it for tests only. The plan leaves it in for safety through migration, which is wise. But after Phase 4 (and after verifying everything in production works without it), we could plan a Phase 5 (outside this doc's scope) to remove the store and its signals, simplifying the code further.
 - Performance: After migration, we might profile certain operations (the document's metrics suggest they did). If we see any lag in point-heavy operations (like selecting thousands of points now triggers maybe a bigger update), we could optimize (e.g., `ApplicationState` could have fine-grained signals if needed, but that reintroduces complexity). Given the test metrics, likely it's fine.

Validated Items vs Discrepancies Recap:

- *Data flow:* Assumption of no circular update **validated** (code matches doc analysis).
- *CurveDataFacade methods:* Exist and match described use; **discrepancy:** missing `ApplicationState` counterparts (to be added) for add/remove – plan assumed they were present.
- *Method signatures:* `update_point` signature in facade corrected by plan and confirmed in code (no status arg) [GitHub](#).

- *Signal handlers*: 8 identified, all found in code at noted locations; signatures and roles as expected. Plan's count correction is **valid**. No missing handlers beyond these.
- *__default__ usage*: Found in code in StateSync, Widget, StateManager, etc., exactly where plan targets removal. Removal plan is **feasible** and addresses all these occurrences (no unexplained usages left).
- *Migration feasibility*: Overall, the plan is comprehensive and aligns with code structure. The only notable **missing implementation** is the ApplicationState point-level mutations, which we will implement. Other assumptions (like existence of `get_all_curve_names()`, `curves_changed` signal, etc.) are all correct – those methods/signals exist in ApplicationState (e.g., `get_all_curve_names` exists [GitHub](#), `curves_changed` is defined [GitHub](#)). The codebase is in a good state to undergo the planned refactoring.

6. Conclusion and Suggestions

Before proceeding with implementation, we recommend addressing the minor discrepancies: implement the missing ApplicationState methods (`add_point`, `remove_point`) and update any code references accordingly. Clarify the Phase 0 status – if some of these were implemented in a branch, merge them or implement afresh. Additionally, ensure that after migration, tests are updated for the new signal signatures and that any references to "`__default__`" (in tests or tools) are removed or handled.

The migration architecture appears **feasible and well-supported by the code**. The careful, phased approach (with an adapter and incremental handler updates) is appropriate given the code's current dual-store setup. Our review finds the plan is quite thorough – discrepancies were minimal and have been noted. By following the document's steps and our recommendations, the transition from `CurveDataStore` to `ApplicationState` as sole source of truth can be completed with minimal risk.

Improvements & Clarifications:

- When implementing `ApplicationState.remove_point`, consider how to handle the case of removing a point that doesn't exist or handling index vs frame logic. Returning a bool as in the plan is good for error checking (the facade uses it to log a warning on failure). Also, ensure to emit `curves_changed` after removal so the UI updates. The plan's omission of how selection is adjusted in ApplicationState during removal should be accounted for (e.g., if a point is removed, should we call `set_selection(curve, new_selection)` to drop that point from selection? Or trust that since selection is by frame, if the frame is gone it's not selected? This depends on whether frame values are unique identifiers – typically they are frame numbers which could be reused if a point is re-added. It might be simplest to clear or update the selection set for that curve on removal to be safe). Document this behavior for consistency.
- It could be helpful to document explicitly in code when we remove the `__default__` fallback that single-curve mode is now handled by always naming curves (for instance, we might automatically name a curve "`default`" (without underscores) or the file name on file open, instead of a hidden default). This isn't strictly necessary but would avoid any confusion for future maintainers wondering how single-curve is identified. The current plan assumes the concept just goes away, which is fine as long as at least one curve name is always present when data is loaded.

- After Phase 4, consider removing or refactoring **StoreManager** and **CurveDataStore**. If it's truly not used, having it around could confuse new developers. If it must stick around until v2 for plugin compatibility or such, maybe clearly mark it as deprecated (even at runtime log a warning if something tries to use it unexpectedly). The plan leaves removal for a future major version, so at least update comments to reflect it is deprecated and not used in core flows.

Overall, the plan is **validated** by our findings. With the above adjustments and careful execution, the ApplicationState migration can be completed successfully. The code is ready for Phase 3 and 4 with only minor enhancements needed, and no fundamental blockers were discovered. The end result will be a cleaner, more robust state management architecture with a single source of truth, which the codebase is evidently prepared for.



Sources