ChatGPT

Asdfghjkl Codebase Review

Feature Progress vs. PLAN.md

According to **PLAN.md**, Asdfghjkl's core features are largely implemented, with a few parts still in progress. Below is a summary of key features compared to the plan:

- **Grid Refinement & State Machine: Implemented.** The core logic for dividing the screen into a 4×10 grid and refining the target region with each keystroke is complete. Classes like `GridLayout`, `OverlayState`, and `OverlayController` match the plan's design. For example, `GridLayout` defines a QWERTY-based 4×10 key mapping and subdivision logic [1] [2]. The `OverlayController` holds the current grid rectangle and updates it on key presses [3], exactly as outlined in the plan.

- **Double-Cmd Activation (Input Layer): Implemented.** A double-press of the Command key toggles the overlay on/off. The code uses a global CGEvent tap and a `CommandTapRecognizer` to detect double-taps, following the plan's state-machine logic (tracking `cmdDown`, timing, and modifier use) [4] [5]. When a double-tap is detected, it calls `overlayController.toggle()` on the main thread [6] [7]. This aligns with the intended behavior: single Cmd presses are ignored, Cmd+Key combos mark "modifier use," and a quick second tap within 0.35s triggers the overlay.

- **Overlay Windows (Visual Overlay Layer): Partially implemented.** Borderless overlay windows are created for **each display**, covering the screen with a translucent grid when active (matching the multi-screen design) [8] [9]. The overlay draws grid lines and highlights the currently selected tile [10] [11]. However, as of now **tile labels** (displaying the letter for each grid cell) are not drawn – the plan noted labels as optional [12], and the implementation currently omits them. The overlay correctly appears on all monitors and highlights only the "active" screen's region (others are dimmed with no highlight). Selecting a different monitor requires moving the mouse and re-triggering the overlay (the plan mentioned possibly adding a "jump display" feature later [13], which isn't implemented yet).

- **Zoom "Loupe" Window: Basic stub.** A floating zoom window is present but not fully functional. The `ZoomWindowController` creates a small borderless window (240×180) that centers on screen [14] [15]. It contains a SwiftUI `ZoomPreviewView` which currently **only displays the size and coordinates** of the target region, not an actual magnified image [16] [17]. The plan's snapshot-based zoom feature (using `CGWindowListCreateImage` to show a magnified live preview) is noted as a future enhancement [18] [19]. The code even includes a `TODO` in `ZoomController.update()` for rendering a magnified snapshot later [20]. In short, the zoom infrastructure is in place (window, published rect, etc.), but capturing and displaying the screen content remains to be done.

- **Mouse Movement & Click (Action Layer): Implemented.** Pressing **Spacebar** moves the mouse to the refined target point and performs a click, then exits the overlay. This uses `SystemMouseActionPerformer`, which warps the cursor and posts a left mouse down/up event via `CGEvent` [21] [22]. After the click, the overlay state is set to inactive and windows are

hidden [23] [24]. This behavior aligns with the plan's step-by-step clicking sequence [25]. The code properly stops consuming keyboard events once the overlay deactivates, returning control to the user's normal workflow.

Overall, the **core functionality is about 80-85% complete** (as the plan itself estimates [26]). The grid navigation, activation gesture, and clicking work as intended. Remaining work mainly involves finishing the UI aspects (the zoom image, any polish on the overlay view) and some system integration tasks (detailed below).

Alignment with Plan and Notable Gaps

By and large, the implementation follows the design outlined in **PLAN.md**. The high-level architecture (input layer, overlay layer, action layer) is reflected in code, and most components are present. However, there are a few inconsistencies or gaps between the plan and the current code:

- **Global Event Tap:** The plan noted that installing the CGEvent tap was a to-do (stubbed out) [27], but in the code this is actually fully implemented. In `InputManager.start()`, a session event tap is created to intercept `flagsChanged` and `keyDown` events [28]. This suggests the implementation progressed further than the last plan update. The event tap callback properly distinguishes Command key changes vs. normal key presses and either toggles the overlay or routes keys to the overlay controller [29] [30]. In practice, this means the app already runs in the background capturing keystrokes (with the required privacy permissions).

- **Overlay State Management:** The plan proposed a single `OverlayState` struct holding `activeScreen`, `currentRect`, `depth`, and `isVisible` [31] as the single source of truth. The implementation simplified this. The `OverlayState` struct in code only tracks whether the overlay is active and the current/root grid rectangles (no explicit screen or depth fields) [32]. The active screen's bounds are managed outside (the app determines the screen under the cursor on activation and uses that) [33] [34]. Depth (number of refinements) isn't stored at all; it can be inferred if needed, but currently is not used. This simplification doesn't break functionality, but it's a slight deviation from the spec. All other aspects of state (current target rectangle, active flag) are handled as planned.

- **Overlay Activation on Multiple Displays:** As designed, the overlay appears on all screens simultaneously [35]. The implementation achieves this by creating an `OverlayWindowController` for each `NSScreen` [36] [37]. However, **choosing the active display** is done implicitly: when toggling on, the app picks the screen under the mouse pointer (or the main screen) as active [38] [33]. All input refining is then constrained to that screen's area. The plan mentioned possibly adding a way to jump between screens via a key later [39], which is not yet implemented. In practice, if the user needs to target another monitor, they must move the cursor and toggle the overlay again – a limitation to address in future iterations.

- **Visual Details of the Overlay:** The code's overlay rendering meets the basic requirements (grid lines, tinted background, highlight box) [40] [11]. Some polish items from the plan are not done yet: for instance, drawing the letter labels in each grid cell (to aid the user in choosing keys) is omitted – currently the overlay is unlabeled, relying on the user's mental 4×10 keyboard map. This is a minor usability gap. Additionally, the plan suggested using the accent color for highlights and a slight animation; the implementation does use `Color.accentColor` for the highlight outline and animates changes smoothly [41] [42], which is in line with best practices and the intended design.

- **Zoom Window Behavior:** As noted, the zoom window does not yet show a magnified screen content. Moreover, its positioning is fixed to screen center on creation, rather than following the cursor or target region. The plan allowed for a simplified v0 implementation (or skipping zoom) [43], so this isn't a critical deviation. It is clear from a `TODO` comment that adding actual zoom imagery is planned [20]. One detail to consider is window layering: the overlay windows use `.screenSaver` level to sit above all content [9], whereas the zoom window is created at `.floating` level [44]. This might cause the zoom window to appear **behind** the full-screen overlay windows. In testing, if the zoom preview is not visible when the overlay is active, the zoom window's level may need to be raised to match the overlay (or added as a child of one overlay window). This is a small inconsistency to verify when finishing the feature.

- **Permission Handling and Feedback:** The implementation currently prints an error if the CGEvent tap cannot be created (likely due to missing Input Monitoring permissions) [45], but does not otherwise alert the user. The plan noted that the app would require Accessibility/Input Monitoring permissions [46]. Improving this by detecting lack of permissions and guiding the user (e.g. via an alert dialog or instructions) is not done yet. This is an important gap for real-world use: without proper handling, a new user might double-tap Cmd and see nothing happen if the app isn't authorized, with no explanation apart from a console log.

In summary, **there are no major functional deviations** – the implemented behavior is very much in line with the plan. The gaps are largely about completeness and polish: finishing the zoom functionality, adding optional UI niceties, and ensuring the app handles edge cases (like permissions or multi-monitor ergonomics) gracefully.

Architectural Evaluation

Code Structure and Modularity

The project is well-structured into a Swift Package with separate modules for core logic and the app UI. The `Package.swift` defines a library target **AsdfghjklCore** (the core logic, which is platform-neutral where possible) and an executable target **Asdfghjkl** for the macOS app [47]. This separation enforces modularity: the core grid navigation and state machine can be built and tested independently of the UI. It's a sound approach that follows Swift best practices for separation of concerns.

Key architectural choices and their merits:

- **Core Logic Isolation:** Classes like `OverlayController`, `GridLayout`, `OverlayState`, `InputManager`, etc., reside in AsdfghjklCore and contain no SwiftUI or AppKit code (guarded by `#if os(macOS)` where needed). This means business logic (e.g. how the grid is refined, how double-tap is detected) can be unit-tested without launching the app UI [48] [49]. It also means the core could potentially run in a command-line mode or on other platforms. In fact, the package includes a minimal CLI `main` for non-macOS (or demo usage), which exercises the overlay logic purely in console output [50] [51]. This decoupling is executed well.

- **Dependency Injection and Testability:** The design makes good use of dependency injection to improve testability and flexibility. For example, `OverlayController`'s initializer accepts a `screenBoundsProvider` closure, a `ZoomController` (optional), and a `MouseActionPerforming` strategy [52]. In production, these default to the real screen size, the live zoom model, and the system mouse action performer. In tests, they can be injected with custom values or stubs. The test suite takes advantage of this to simulate different screen sizes

and to verify that `OverlayController` calls the mouse-click action with the correct coordinates [53] [54]. Similarly, the `CommandTapRecognizer` accepts a custom time source for testing double-tap timing [55] [56]. These are signs of a thoughtful architecture that values correctness and maintainability.

- **State Management:** The app maintains clear separation between the *source of truth* for the overlay state and the *presentation state*. The `OverlayController` owns the authoritative state (`OverlayState`) inside the core module. When the state changes, it notifies observers via a closure (`stateDidChange`) [57]. The App module uses this to update an `OverlayVisualModel` – an `ObservableObject` that SwiftUI views observe [58] [59]. This way, the SwiftUI overlay views react to changes in `OverlayVisualModel` (published properties like `isActive` and `currentRect`) while the core logic remains UI-agnostic. This is a robust pattern, ensuring the UI is always in sync with the underlying state without duplicating business logic. One minor critique is that it introduces a bit of duplication (the `OverlayVisualModel` mirrors fields from `OverlayState` [59]). Alternatively, `OverlayState` itself could have conformed to `ObservableObject` with `@Published` properties, but that would have pulled in Combine frameworks into core. The chosen approach keeps core lean and pure, at the cost of an extra model layer – a reasonable trade-off.

- **Event Handling and App Lifecycle:** Using an `NSApplicationDelegateAdaptor` in SwiftUI App structure is a good choice to handle AppKit specifics in `AppDelegate` [60] [61]. The `AppDelegate` sets up the controllers on launch and tears down on terminate, much like in a traditional AppKit app. There's a clear flow: on launch, create overlay windows for each screen, set up the input manager (which installs the event tap), and prepare the initial screen selection [62] [38]. The design ensures that once the app is running, it doesn't require a visible UI window to function – it's effectively a background utility triggered by the keyboard. This matches the intended use case.

- **Use of Protocols and Abstractions:** The code defines a `MouseActionPerforming` protocol with a default implementation `SystemMouseActionPerformer` for the actual CGEvent posting [63] [24]. This abstraction allowed injecting a `StubMouseActionPerformer` in tests to verify the click coordinates without actually moving the cursor [53] [54]. Such patterns indicate a high-quality architecture. Similarly, the separation of `CommandTapRecognizer` (which has no dependencies) from `InputManager` means the double-tap logic can be tested in isolation [64] [65].

Overall, the architecture demonstrates **strong modularity, low coupling between components, and clear layering** (Input, Overlay, Action layers as described in the plan [66] [67]). Each class has a focused responsibility, and the flow of data (from global events -> input manager -> overlay controller -> visual model -> SwiftUI view -> user) is logical and easy to follow.

Swift Best Practices and Patterns

From a Swift and macOS development standpoint, the implementation follows best practices in several ways:

- **SwiftUI + AppKit Integration:** The project leverages SwiftUI for the overlay rendering while still using AppKit for window management. `NSHostingController` is used to embed SwiftUI views (`OverlayGridView`, `ZoomPreviewView`) in `NSWindow` objects [68] [44]. This is the recommended approach to create overlay windows with SwiftUI content. The code properly

configures window properties (borderless, transparent, ignores mouse events, on all spaces, etc.) in line with Apple guidelines and the plan's recommendations [69] [9]. By marking the overlay windows as `.canJoinAllSpaces` and `.fullScreenAuxiliary`, it ensures the overlay appears even over full-screen apps – exactly as intended for a global utility.

- **Combine/SwiftUI State Publishing:** The use of `@Published` in `OverlayVisualModel` and `ZoomController` allows SwiftUI views to automatically update when the state changes [59] [70]. This reactive update mechanism is idiomatic SwiftUI. Moreover, the code carefully uses `DispatchQueue.main.async` when updating state in response to background events (the event tap) [57] [58]. This ensures UI state changes occur on the main thread, preventing any threading issues with SwiftUI's state updates. Such attention to thread correctness is important and well-handled here.

- **Memory Management and Safety:** The implementation avoids retain cycles by using `[weak self]` in closures that capture `self` (e.g. the `stateDidChange` callback and InputManager's onDoubleTap closure) [57] [7]. This prevents potential memory leaks where long-lived closures (like the event tap callback) reference the AppDelegate or controllers. Additionally, the code is careful to keep the CGEvent tap alive by storing the `CFMachPort` and run loop source as properties of `InputManager` [71] [28]. This prevents the tap from being garbage-collected. On termination, the app currently doesn't explicitly remove the event tap, but since the process exits and the tap is tied to the run loop, this is acceptable. (For completeness, one could disable the tap in `applicationWillTerminate`, but it's a minor point.)

- **Coding Style and Clarity:** The code is concise and readable. It uses Swift's modern features (e.g., guard, defer, computed properties) appropriately. For instance, `CommandTapRecognizer.handleCommandUp` uses `defer` to ensure state is reset regardless of branch [72]. Optionals are handled safely (e.g., checking for valid characters from events, optional CGEvent creation). The team also gave meaningful names to variables and functions, making the code self-documenting in many places. Inline documentation is light but sufficient, given that the PLAN.md and README.md provide higher-level context. Key functions like `InputManager.handleKeyDown` have doc comments describing their purpose [73].

- **Adherence to Plan/Spec:** It's worth noting that the developers treated the plan as a spec and followed it closely. This disciplined approach is evident in one-to-one correspondences (for example, the double-tap algorithm in code mirrors the plan pseudocode step by step [74] [5]). Such consistency is a positive sign: the engineers kept the implementation aligned with design, which will make maintenance easier since the documentation (plan) actually matches the code behavior.

In summary, the architecture is **sound and implemented with best practices in mind**. The codebase is modular, testable, and uses SwiftUI/AppKit appropriately. There is a clear separation of concerns and a clean flow of data and control. It demonstrates a good balance between simplicity and flexibility.

Testing, Documentation, and Quality

The project exhibits a commendable focus on quality through testing and documentation:

- **Unit Testing:** A comprehensive suite of unit tests covers the core logic. There are tests for grid layout calculations, overlay state transitions, and the command tap recognizer, among others. For example, `GridLayoutTests` verify that pressing keys yields the expected grid subdivisions

[75] [76], and `OverlayControllerTests` simulate key sequences to ensure the overlay refines and clicks correctly [77] [49]. The `CommandTapRecognizerTests` cover edge cases of the double-tap timing and modifier logic [56] [78]. These tests give confidence that the core mechanics work as intended and also serve as living documentation of expected behavior. The use of test doubles (stubs) for things like mouse actions confirms that side effects are correctly invoked [53] [54]. One area untested (understandably) is the integration of the CGEvent tap and the SwiftUI views, which would require UI/system testing. Still, the critical logic is well-validated.

- **Continuous Integration:** The README mentions GitHub Actions workflows for testing and building [79]. On each push or PR, the tests are run on macOS with Swift 6.2 [80]. There is also a workflow to build a release binary and upload it as an artifact. This CI setup indicates a mature development process, catching regressions early and producing deliverables for testers or users to try. It's a best practice to have automated tests and builds, especially for a tool that interacts with system events, where manual testing every scenario can be tedious.

- **Documentation & Planning:** The existence of **PLAN.md** itself (a "master plan" document) is a strong positive. It provides a clear specification of the intended features, architecture, and even an implementation roadmap [81] [82]. Maintaining this alongside the code helps new contributors (or future maintainers) understand the purpose of each component. The inline references in AGENTS.md to update docs and tests as you go [83] suggest the project might have been developed in tandem with an AI assistant or simply with an eye towards keeping documentation up-to-date. The README is concise but sufficient: it explains what the app does and how to build/run it [84] [85]. It even acknowledges that the package "now ships a SwiftUI/AppKit macOS app lifecycle" and describes the current state of global event capture and overlay windows [86]. This honesty in documentation (explaining what works and what is placeholder) is excellent for a reader or user. For instance, a user will know from the README that the overlay is triggered by double-tap Cmd and what to expect when refining and clicking.

- **Code Comments and Clarity:** While many functions lack extensive comments, their names and the plan coverage make their intent clear. Critical or non-obvious behaviors do have comments (e.g., the `handleEvent` switch in InputManager has a note to re-enable the tap if it's disabled by the system [29]). The use of Swift's expressive syntax reduces the need for redundant comments. In general, the code formatting and naming follow Swift conventions, which aids readability.

Given the above, the project appears to prioritize correctness and maintainability. The combination of up-to-date documentation, thorough unit tests, and CI means new changes can be made with confidence and reviewed against the original plan. It's evident the developers aimed for a high-quality, reliable implementation.

Suggestions for Further Development

With the foundation in great shape, there are a number of next steps and enhancements that would improve the product and address remaining gaps:

- **Complete the Zoom Functionality:** Finishing the zoom/loupe feature is a top priority for feature parity with the plan. This entails capturing a magnified image of the target area and displaying it in the `ZoomWindowController`. As suggested in PLAN.md, using `CGWindowListCreateImage` to grab the screen content of `currentRect` and show it scaled up would fulfill this. The architecture is ready for it – e.g., implement the

`ZoomController.update(rect:)` TODO [20] to produce an NSImage or CGImage, and show that in the `ZoomPreviewView` instead of the placeholder rectangle. This will greatly help users hit small targets. Additionally, consider positioning the zoom window near the cursor or target point rather than always centering it. This way, the user's eye doesn't have to move far. (This could be as simple as moving the window's center to the target `GridRect` location when updating, with some clamping to keep it on-screen.)

- **Overlay Usability Tweaks:** Adding **grid cell labels** would make the overlay more user-friendly. Right now, users must intuit which key corresponds to which tile. Drawing a small character (e.g., "Q", "W", "E"... or "1", "2"... in the cells) can reduce cognitive load. The plan mentioned this as an option [87]. The implementation can leverage SwiftUI: for example, overlay a text label in each grid cell in `OverlayGridView.gridLines` path or as a ZStack with separate Text views. This can be done conditionally or with a subtle style so as not to clutter the view. Even if not all keys are labeled (since some non-alphanumeric keys like ";" or "," are included), labeling most tiles would help. If not in the initial version, perhaps behind a setting toggle for advanced users.

- **Multi-Display Enhancements:** Right now, multi-monitor setups are supported in that the overlay appears on all screens, but the interaction is limited to the one "active" screen (usually where the cursor was at activation). In the future, you might implement a way to quickly switch the active screen **without** restarting the overlay. For instance, pressing a function key or a certain modifier while overlay is active could jump the highlight to another display. (This could be implemented by mapping a key to cycle through `NSScreen.screens` and updating `activeScreenRect` accordingly.) This wasn't strictly required by v1, but it was hinted as a later improvement in the plan [39]. It would make the tool more seamless in multi-monitor workflows.

- **Robust Permission Handling:** Improve how the app handles required macOS permissions. On first launch, if the event tap fails due to lack of privileges, the app should clearly communicate this. Since macOS does not allow auto-prompting for Input Monitoring, the app could at least detect the failure and pop up a dialog instructing the user to enable the permission in System Settings (Security & Privacy > Privacy > Input Monitoring and Accessibility). The current `print("Failed to create CGEvent tap…")` [88] will be invisible to users running the app normally (outside of a debugger). Providing user-facing feedback will prevent confusion when nothing happens on double-tap due to missing permissions. Additionally, the documentation (README) can include a note about granting these permissions.

- **Application UX & Controls:** Consider providing a minimal UI to manage the overlay's lifecycle. For example, running as a true menu bar app (LSUIElement) with an icon could allow users to quit or toggle the service easily. Right now, since the app has no menu or window, the only way to quit might be via Activity Monitor or by creating your own mechanism (unless a user knows to press Cmd+Q when the app is focused, which may not be obvious if no dock icon exists). A menu bar item or a small preferences window could also allow customization (like adjusting the double-tap threshold, the overlay color/opacity, etc., in the future). Even without going full GUI, a global hotkey to turn the overlay on/off (besides the double Cmd) might be useful, but that enters redundancy – the double Cmd is a good choice as is.

- **Error Handling and Edge Cases:** Continue to harden the app by handling edge cases. For example, what happens if the user tries to use the overlay while a secure text field is focused (macOS might suppress key events for security)? Or if the user has an international keyboard where keys might not map one-to-one with the QWERTY layout provided – perhaps allow customization of the keymap if needed. Also, test what happens if the user triggers the overlay,

then *very quickly* triggers it again (the code toggles state on double-tap of Cmd – a very fast quadruple-tap might toggle on then off immediately). The logic seems to handle it by toggling state each time, which is fine. These are minor, but worth verifying. Another scenario: if multiple monitors have very different resolutions or scaling, ensure the grid mapping still works accurately (the code uses raw screen coordinates, which should be fine since it operates in full-resolution coordinates).

- **Performance Considerations:** The current implementation should be very lightweight – it only reacts to key events and draws simple SwiftUI shapes. Once the zoom snapshot feature is added, keep an eye on performance. Capturing the screen can be expensive if done too often. The plan's guidance to only update on each key press (not continuously) is wise [89]. Implement caching or rate-limiting if necessary (though keypresses are human-rate, so likely fine). Also, using `.orderFrontRegardless()` for windows is appropriate to ensure they appear. Just ensure that hiding/showing the overlay windows (and creating them) on each activation is flicker-free. The code currently retains the windows after first creation (storing them in the controller) and reuses them, which is good for performance [90].

- **Accessibility & Inclusivity:** Think about how this tool can be used by people with different needs. For example, colorblind users might benefit if the highlight rectangle's color or thickness could be customized for better visibility (the current use of the system accent color with transparency might be hard to spot for some). VoiceOver (screen reader) users likely wouldn't use this tool (since it's very visually oriented), but at the very least the overlay windows are marked to ignore mouse events – it might also be wise to mark them as non-accessible (so VoiceOver doesn't focus them). This might be the default with borderless panels, but it's something to verify. These considerations can be addressed in later versions, but keeping them in mind from the start is beneficial.

- **Additional Testing:** While unit tests are great, as the UI pieces come together, some integration testing would be valuable. For instance, using XCTest UI tests or a simple manual test plan to simulate a full user interaction: double-tap Cmd, press a sequence of letters, press space, etc., across multiple screens. This would help catch any timing issues or focus problems (e.g., does an active overlay prevent all other input as intended? Does pressing Escape reliably cancel in all cases?). Given the low-level nature (CGEvent taps), some of these tests might need to be manual or use assistive automation. Ensuring the overlay doesn't interfere with system behavior beyond its scope is important (the current approach of returning the event or `nil` to propagate or consume keystrokes is correct [30], but one should test, for example, that while overlay is active, those keys truly don't trigger actions in the front app).

- **Documentation & Onboarding:** Once the app is near feature-complete, a brief user guide or in-app onboarding (even as a README or help menu) would be helpful. The current README is developer-focused (build and run). For an end-user release, instructions on enabling permissions and perhaps a visual diagram of how the keyboard maps to the screen might be useful. Even a tiny overlay legend (maybe flashing the sections when activated for the first time) could help users understand the mapping. These are more product considerations, but they elevate the overall experience.

In summary, **most suggestions revolve around completing known to-dos (zoom view, labels), enhancing user experience (feedback, settings), and hardening the app**. The good news is the architecture in place can support all these improvements without major refactoring. The code is clean and modular enough that adding features like a settings UI or additional overlay behavior can be done in isolation, reusing the existing core.

Conclusion

Asdfghjkl's implementation is **well-aligned with the PLAN.md specification and demonstrates strong engineering practices**. The core functionality – using the keyboard to precisely move the mouse cursor – is largely implemented and works as designed. The architecture is clean, separating core logic from UI and leveraging Swift's strengths (Combine, SwiftUI, protocol-oriented design) to create a testable and maintainable codebase. Notably, the development process included upfront planning, unit testing, and continuous integration, which shows a level of rigor often seen in production-ready software.

At this stage, the app feels like a working **MVP (Minimum Viable Product)**. The main missing elements are around polish and completeness of the user interface (the zoom preview imagery and some quality-of-life features). There are no glaring architectural flaws; on the contrary, the code structure will accommodate future changes easily.

From a product perspective, a bit more work is needed to make it seamless for end users – primarily around permission handling and perhaps providing a way to run the app in the background with easy control. Once those are addressed and the remaining planned features are built out, Asdfghjkl will be a compelling utility. It already showcases a clever solution to keyboard-driven mouse control, implemented in a robust way. The development so far has set a solid foundation, and with the suggested improvements, the project will not only align with the plan but likely exceed user expectations in terms of reliability and usability.

Sources:

- Asdfghjkl Project Plan 26 91
- Asdfghjkl Core Implementation (selected excerpts) 3 5
- Asdfghjkl UI and Window Code 9 40
- Asdfghjkl Test Suite and README 49 79

---

1 4 6 8 12 13 18 19 21 25 26 27 31 35 39 43 46 66 67 69 74 81 82 87 89 91 PLAN.md
https://github.com/dave1010/Asdfghjkl/blob/ecc1fcc321a4b1a950f2891000a60a84548b8dce/PLAN.md

2 GridLayout.swift
https://github.com/dave1010/Asdfghjkl/blob/ecc1fcc321a4b1a950f2891000a60a84548b8dce/Sources/AsdfghjklCore/GridLayout.swift

3 23 52 OverlayController.swift
https://github.com/dave1010/Asdfghjkl/blob/ecc1fcc321a4b1a950f2891000a60a84548b8dce/Sources/AsdfghjklCore/OverlayController.swift

5 55 72 CommandTapRecognizer.swift
https://github.com/dave1010/Asdfghjkl/blob/ecc1fcc321a4b1a950f2891000a60a84548b8dce/Sources/AsdfghjklCore/CommandTapRecognizer.swift

7 28 29 30 45 71 73 88 InputManager.swift
https://github.com/dave1010/Asdfghjkl/blob/ecc1fcc321a4b1a950f2891000a60a84548b8dce/Sources/AsdfghjklCore/InputManager.swift

9 68 90 OverlayWindowController.swift
https://github.com/dave1010/Asdfghjkl/blob/ecc1fcc321a4b1a950f2891000a60a84548b8dce/Sources/Asdfghjkl/OverlayWindowController.swift

10 11 16 17 40 41 42 OverlayViews.swift
https://github.com/dave1010/Asdfghjkl/blob/ecc1fcc321a4b1a950f2891000a60a84548b8dce/Sources/Asdfghjkl/OverlayViews.swift

14 15 44 ZoomWindowController.swift
https://github.com/dave1010/Asdfghjkl/blob/ecc1fcc321a4b1a950f2891000a60a84548b8dce/Sources/Asdfghjkl/ZoomWindowController.swift

20 70 ZoomController.swift
https://github.com/dave1010/Asdfghjkl/blob/ecc1fcc321a4b1a950f2891000a60a84548b8dce/Sources/AsdfghjklCore/ZoomController.swift

22 24 63 MouseActionPerformer.swift
https://github.com/dave1010/Asdfghjkl/blob/ecc1fcc321a4b1a950f2891000a60a84548b8dce/Sources/AsdfghjklCore/MouseActionPerformer.swift

32 OverlayState.swift
https://github.com/dave1010/Asdfghjkl/blob/ecc1fcc321a4b1a950f2891000a60a84548b8dce/Sources/AsdfghjklCore/OverlayState.swift

33 34 36 37 38 50 51 57 58 60 61 62 App.swift
https://github.com/dave1010/Asdfghjkl/blob/ecc1fcc321a4b1a950f2891000a60a84548b8dce/Sources/Asdfghjkl/App.swift

47 Package.swift
https://github.com/dave1010/Asdfghjkl/blob/ecc1fcc321a4b1a950f2891000a60a84548b8dce/Package.swift

48 49 53 54 77 OverlayControllerTests.swift
https://github.com/dave1010/Asdfghjkl/blob/ecc1fcc321a4b1a950f2891000a60a84548b8dce/Tests/AsdfghjklTests/OverlayControllerTests.swift

56 64 65 78 CommandTapRecognizerTests.swift
https://github.com/dave1010/Asdfghjkl/blob/ecc1fcc321a4b1a950f2891000a60a84548b8dce/Tests/AsdfghjklTests/CommandTapRecognizerTests.swift

59 OverlayVisualModel.swift
https://github.com/dave1010/Asdfghjkl/blob/ecc1fcc321a4b1a950f2891000a60a84548b8dce/Sources/Asdfghjkl/OverlayVisualModel.swift

75 76 GridLayoutTests.swift
https://github.com/dave1010/Asdfghjkl/blob/ecc1fcc321a4b1a950f2891000a60a84548b8dce/Tests/AsdfghjklTests/GridLayoutTests.swift

79 80 84 85 86 README.md
https://github.com/dave1010/Asdfghjkl/blob/ecc1fcc321a4b1a950f2891000a60a84548b8dce/README.md

83 AGENTS.md
https://github.com/dave1010/Asdfghjkl/blob/ecc1fcc321a4b1a950f2891000a60a84548b8dce/AGENTS.md