Architectural Feasibility Study: Implementing Niri-Style Infinite Scrolling on the Windows Desktop Window Manager

1. Introduction: The Evolution of Spatial Window Management

The graphical user interface (GUI) has, for decades, relied on the metaphor of the "desktop"—a finite, static 2D plane upon which applications are stacked or tiled. This paradigm, while functional for basic tasks, has increasingly strained under the weight of modern multi-tasking workflows involving high-resolution ultrawide monitors and dozens of concurrent application windows. In the Linux ecosystem, a radical departure from this static model has emerged, popularized first by the **PaperWM** extension for GNOME and subsequently refined into a standalone Wayland compositor known as **Niri**. This new paradigm, often termed "scrollable tiling" or "infinite tiling," reimagines the workspace not as a finite screen, but as an infinite horizontal ribbon or strip.

In this model, windows are arranged linearly along an infinite horizontal axis. The physical monitor acts merely as a viewport or camera that slides over this ribbon. When a new window opens, it does not compress existing windows to fit the screen (as in traditional tiling window managers like **i3**, **sway**, or **bspwm**); instead, it simply appends itself to the strip, extending the logical workspace.[1] This approach decouples the logical arrangement of applications from the physical constraints of the display hardware, offering a profound ergonomic advantage: users maintain spatial memory of their applications (e.g., "browser is to the right of the terminal") without sacrificing content legibility to shrinking window sizes.

The query addressed in this report concerns the translation of this specific user experience—the "Niri experience"—to the Microsoft Windows ecosystem. Windows users, particularly developers and power users utilizing ultrawide displays, face a stark lack of native support for this workflow.[4] While tiling window managers for Windows exist, most notably **Komorebi** and **GlazeWM**, they traditionally adhere to the static Binary Space Partitioning (BSP) or stack-based layouts.[5]

This report provides an exhaustive technical analysis of the feasibility of introducing Niri-like infinite scrolling to Windows. It specifically evaluates two divergent engineering pathways: **forking the existing Komorebi window manager** to support infinite scrolling mechanics versus **porting the Niri codebase** directly to the Windows platform. This analysis is grounded

in a deep examination of the Windows Desktop Window Manager (DWM) architecture, the Win32 API constraints regarding off-screen rendering, and the internal Rust architectures of both Niri and Komorebi.

---

2. The Windows Desktop Window Manager (DWM) Environment

To assess the feasibility of either porting Niri or modifying Komorebi, one must first establish a rigorous understanding of the operating environment: the Windows Desktop Window Manager (DWM). Unlike the Linux ecosystem, where the display server (Wayland) and the window manager (compositor) are often unified and replaceable, Windows enforces a rigid separation where the DWM is the exclusive compositor. Any third-party window manager on Windows effectively operates as a "puppet master," manipulating window handles (HWNDs) from the outside, rather than controlling the rendering pipeline itself.

2.1 Historical Context and Architectural Constraints

Introduced in Windows Vista, the DWM represented a fundamental shift from the legacy Stacking Window Manager model of Windows XP. In the XP era, applications painted directly to the video card's front buffer. If a window was "hung" or unresponsive, its visual representation would degrade (e.g., the "trail of windows" effect).[7] The DWM introduced a compositing surface where every application draws to an off-screen buffer (surface), and the DWM aggregates these buffers into a final frame to be sent to the GPU.[8]

For a Niri-like implementation, this architecture presents a critical "impedance mismatch." Niri, as a Wayland compositor, owns the rendering loop. To "scroll" the infinite strip, Niri simply translates the coordinates of the texture composition. It is a graphical operation, inherently smooth and tear-free. On Windows, however, a window manager like Komorebi cannot instruct the DWM to "render this window texture at coordinate $X + 10$." Instead, it must use the Win32 API SetWindowPos to instruct the Operating System to move the window's logical position. The OS then processes this request, sends a WM_WINDOWPOSCHANGING message to the application, waits for the application to respond (potentially), and then the DWM composites the window at the new location.[9]

This indirect control mechanism introduces significant latency and computational overhead. Achieving the 1:1 fluid tracking of a touchpad gesture—a hallmark of Niri [11]—is exceptionally difficult on Windows because the "move" operation is not a lightweight texture translation but a heavyweight OS window state change.

2.2 The Virtual Screen Coordinate System

A primary feasibility question is whether Windows supports the concept of an "infinite" workspace. Windows combines all connected physical monitors into a single logical coordinate system known as the **Virtual Screen**.[12] The primary monitor typically originates at $(0, 0)$, with secondary monitors extending into positive or negative coordinates depending on their arrangement.

2.2.1 Coordinate Limits and Overflow

The Win32 API uses 32-bit signed integers for coordinates, theoretically allowing values between $-2,147,483,648$ and $+2,147,483,647$. In practice, however, the "interactive" desktop coordinate space is far more constrained. Historical limitations in the Graphics Device Interface (GDI) often limited coordinates to 16-bit values ($\pm 32,767$). While modern Windows 10/11 DWM handles larger coordinates better, "ghosting" or rendering bugs can occur when windows are pushed extremely far off-screen.[13]

For an infinite scrolling implementation, this means a "true" infinite strip is technically risky. A window manager must likely implement a "circular buffer" or "coordinate renormalization" strategy. As the user scrolls right, effectively increasing the X coordinate to $50,000$ or $100,000$, the manager might eventually need to shift the entire coordinate system back to zero to avoid hitting legacy overflow boundaries or precision errors in floating-point calculations used for animations.[9]

2.3 Off-Screen Rendering and Occlusion Culling

One of the most significant barriers to porting the Niri experience is the behavior of the DWM regarding off-screen windows. In Niri, a window that is effectively "off-screen" (outside the viewport) is simply not composited, but its surface remains active in memory.

On Windows, the DWM employs aggressive **Occlusion Culling** to optimize resource usage. If a window is fully obscured by another window or moved completely outside the bounds of the virtual screen, the DWM may effectively "sleep" the rendering of that window.[14]

- **The "Black Rectangle" Phenomenon:** If a user rapidly scrolls the infinite strip, bringing an off-screen window into the viewport, there is a non-zero probability that the window will appear as a black rectangle or display stale (outdated) content for several frames until the DWM prioritizes its repainting.
- **Process Priority Degradation:** Windows considers off-screen applications as background tasks. The OS scheduler may reduce the CPU priority of these processes to save power, leading to sluggish response times when they are suddenly scrolled back into view.[15]

This behavior necessitates a complex workaround in any Windows implementation: the

window manager cannot simply move windows to $X = 50000$. It must likely employ a technique such as keeping a 1-pixel sliver of the window visible on the edge of the virtual screen or utilizing the **Cloaking** API (DwmSetWindowAttribute with DWMWA_CLOAK) to hide the window while signaling to the OS that it should remain active.[16]

2.4 API Capabilities: SetWindowPos vs. Visuals

The primary tool for any Windows window manager is SetWindowPos. However, this API is synchronous and heavyweight.

- **BeginDeferWindowPos / EndDeferWindowPos:** To minimize the "tearing" effect where windows move sequentially rather than simultaneously, Windows provides the DeferWindowPos structure. This allows the window manager to batch multiple move commands into a single OS call.[13] This is absolutely critical for implementing the scrolling mechanism; without it, scrolling a strip of 5 windows would result in a visual cascade of movement rather than a unified slide.
- **DwmSetWindowAttribute:** This API allows for modifying the window's integration with DWM, including rounded corners (DWMWA_WINDOW_CORNER_PREFERENCE) and border colors. However, these are static attributes. Unlike Niri, which can apply dynamic shaders to windows during the scroll animation (e.g., blurring or bending), the Windows DWM provides no public API for injecting custom shaders into the composition pass.[2]

---

3. Niri Architecture Analysis

To evaluate the feasibility of a port, we must deconstruct Niri not just as software, but as a system of dependencies. Niri is built in Rust, leveraging the safety and concurrency features of the language, but its architectural dependencies are inextricably linked to the Linux graphics stack.

3.1 The Wayland Compositor Model

Niri is built upon the **Smithay** library, a Rust building block for Wayland compositors.[17] This dependency stack is the primary barrier to a Windows port.

- **Input Handling:** Niri utilizes libinput to handle raw hardware events from mice, keyboards, and touchpads. This library provides the high-fidelity gesture data (e.g., 1:1 touchpad tracking) that makes Niri's scrolling feel physical and responsive. Windows does not use libinput; it uses the Raw Input API and WM_INPUT messages, which have entirely different data structures and behaviors.
- **Rendering Backend:** Niri uses wgpu (WebGPU for Rust) or direct OpenGL calls to render the scene graph to a DRM/KMS (Direct Rendering Manager / Kernel Mode Setting) surface. It owns the "scanout" buffer. On Windows, the "scanout" is owned exclusively by dwm.exe. There is no mechanism for a user-space application to take over the scanout

buffer without writing a custom kernel-level display driver (comparable to IddSampleDriver).

3.2 The Infinite Strip Layout Engine

Despite the backend incompatibilities, Niri's *logic* is a significant asset. The core innovation of Niri is its **Layout Engine**—the algorithmic module that determines *where* windows should be placed given a set of constraints (window count, window sizes, viewport position).

- **Decoupled Logic:** Snippets indicate that Niri's layout logic is separated from its rendering loop.[19] Concepts like "center focused column," "fractional scaling rounding," and "dynamic gap calculation" are mathematical operations implemented in Rust structs.
- **Coordinate Management:** Niri maintains a logical coordinate system that is independent of the physical pixel grid until the final composition step. This allows it to handle fractional scaling (e.g., 150% scale) by performing layout calculations in floating-point logical pixels and only rounding to integers at the render stage.[20]

This separation of concerns suggests that while the *application* Niri cannot run on Windows, its *brain*—the layout algorithm—is theoretically portable. A Windows implementation could import Niri's layout crate (or a fork of it) to calculate the target rectangles for windows, while using a different backend (Win32) to actually apply those rectangles.

3.3 State Management and Persistence

Niri treats workspaces as dynamic entities. Unlike static TWMs where workspaces are often pre-defined buckets, Niri's workspaces are created and destroyed dynamically as windows are added or removed.[2] It persists the "view position" of each workspace independently. This state management logic is also platform-agnostic Rust code, managing vectors of Window structs. This creates a strong case for code reuse if the platform-specific Window handle can be abstracted.

---

4. Komorebi Architecture Analysis

Komorebi represents the state-of-the-art for tiling window management on Windows. Also written in Rust, it is designed specifically to navigate the constraints of the Win32 API. Analyzing its architecture reveals both the foundation it offers for an infinite scrolling fork and the limitations of its current implementation.

4.1 Core Architecture: komorebi-core vs. komorebi

Komorebi's codebase is bifurcated into two primary crates:

1. **komorebi-core:** This crate contains the platform-agnostic logic. It defines data structures like WindowManager, Monitor, Workspace, and Container.[22] It handles the

layout algorithms (BSP, Stack, Grid). Crucially, this is where the layout math resides.
2. **komorebi (Binary):** This crate handles the Win32 specifics. It manages the event loop, hooks into Windows events (via SetWindowsHookEx or WinEventHook), and executes the SetWindowPos calls mandated by the core logic.

This separation mimics the architecture required to port Niri's logic: komorebi-core effectively acts as the "Layout Engine" for the Windows implementation.

4.2 Existing Scrolling Layout Implementation

The research snippets confirm that Komorebi *already* possesses a layout variant explicitly named Scrolling.[24]

- **Mechanism:** The current implementation appears to be a basic "multi-column" view. The user specifies a scrolling-layout-columns count (e.g., 2), and Komorebi renders 2 columns on screen.
- **Limitations:**
  - **Fixed Widths:** Unlike Niri, which allows dynamic resizing of individual columns in the strip, Komorebi's current scrolling layout appears to force uniform widths or simple splitting.[26]
  - **Focus Bugs:** Snippet [27] highlights a critical bug: when changing focus in a single-column scrolling layout, the "view" does not scroll to the new window automatically if transparency is enabled. This indicates that the "camera follow" logic is brittle or improperly synchronized with the visual state.
  - **No "Infinite" State:** The current implementation likely does not maintain a virtual coordinate system extending far beyond the monitor. It likely re-shuffles windows into the visible slots rather than simulating a camera panning over a static strip. This results in a "jumping" effect rather than a "scrolling" effect.

4.3 Input Handling and Integration

Komorebi relies on **whkd** (Windows Hotkey Daemon) or **AutoHotKey** for input management.[28] It does not natively handle raw mouse or touchpad gestures for navigation. This is a significant deviation from Niri. In Niri, a 3-finger swipe is processed internally to adjust the viewport offset. In Komorebi, a swipe must be translated by an external tool into a CLI command (e.g., komorebic focus left), which creates discrete, stepped movement rather than analog fluid scrolling.

---

5. Feasibility Study 1: Direct Porting of Niri

The option of "Porting Niri" implies taking the existing YaLTeR/niri repository and modifying it to compile and run on Windows. The analysis below breaks down why this is structurally

infeasible as a direct port.

5.1 The Dependency Chain Failure

A Rust project is defined by its Cargo.toml. Niri's dependency tree is rooted in Linux-specific technologies:

- **smithay:** This crate abstracts Wayland protocols and backend initialization (DRM/KMS, libinput). There is no Windows equivalent for Smithay. To port Niri, one would effectively have to write "Smithay for Windows," a project of immense scope that involves wrapping the entire Win32 Windowing API in a safe Rust abstraction that mimics Wayland's compositor API.[17]
- **udev:** Niri uses udev for device discovery (hotplugging monitors and input devices). Windows uses PnP (Plug and Play) and WM_DEVICECHANGE messages. The logic is fundamentally different.
- **nix:** A crate for *nix system calls. Niri uses this for everything from file descriptors to socket management (for IPC).

5.2 The Rendering Barrier

As established in Section 2, Niri is a renderer. It owns the pixels. To port Niri to Windows, one would have to strip out 100% of the rendering code because dwm.exe owns the pixels.

- **Consequence:** A "ported" Niri would no longer be Niri. It would be a logic engine that sends commands to SetWindowPos. It would lose the rounded corners, the shadows, the shader-based animations, and the perfect frame synchronization that defines the Niri user experience.

5.3 Verdict: Code Re-Use, Not Porting

It is technically impossible to "port" Niri in the sense of compiling the application for Windows. The only feasible path involving Niri is to treat it as a **Reference Implementation**. A developer can read Niri's source code to understand the *algorithms* for infinite scrolling (e.g., how to handle gaps, struts, and fractional scaling in a scrolling context) and re-implement those algorithms in a Windows-native harness.

---

6. Feasibility Study 2: Forking and Extending Komorebi

This option involves taking the existing Komorebi codebase and modifying its Scrolling layout logic to mimic Niri's behavior. This is the **recommended path** due to the high reusability of Komorebi's Win32 boilerplate.

6.1 The "Virtual Strip" Implementation Strategy

The primary enhancement required for Komorebi is to decouple the "Workspace" from the "Physical Monitor."

- **Current State:** In komorebi-core, a Workspace generally assumes its dimensions match the Monitor it is displayed on.
- **The Fork:** The fork must introduce a concept of VirtualWidth. In a scrolling layout, VirtualWidth is the sum of the widths of all windows plus gaps.
    - **Camera Logic:** The workspace must maintain a scroll_offset (float).
    - **Layout Function:** The calculate_layout function, which currently returns a list of Rects visible on the monitor, must be updated. It should calculate Rects for *all* windows in the strip based on scroll_offset.
        - If Window_X + Window_Width < 0 (Left of screen): Mark as **Off-Screen Left**.
        - If Window_X > Monitor_Width (Right of screen): Mark as **Off-Screen Right**.
        - Otherwise: Calculate the visible intersection.

6.2 Managing the "Off-Screen" Windows (The Cloaking Solution)

To solve the DWM occlusion and "black rectangle" issues (Section 2.3), the fork must implement a hybrid visibility strategy using **Cloaking**.

- **Why not SW_HIDE?** Standard hiding removes the window from the taskbar and Alt-Tab order, causing confusion.
- **Why DwmSetWindowAttribute?** Using DWMWA_CLOAK allows the window to remain "present" (in Alt-Tab and Taskbar) but not rendered.
- **The "Buffer Zone" Strategy:** To ensure smooth scrolling, the fork should not cloak *immediately* at the edge of the screen. Instead, it should maintain a "Buffer Zone" of perhaps 1000 pixels on either side of the viewport.
    - Windows in the **Viewport**: Fully managed via SetWindowPos.
    - Windows in the **Buffer Zone**: Positioned off-screen via SetWindowPos but kept visible (uncloaked) to ensure DWM keeps their surface buffer fresh.
    - Windows **Beyond Buffer**: Cloaked to save resources.

6.3 Implementing Fluid Animation (The "Jank" Mitigation)

Komorebi currently supports animations for window creation/destruction, but panning animations are harder.

- **The Feedback Loop:** Smooth scrolling requires an animation loop running at the monitor's refresh rate (e.g., 144Hz).
- **Implementation:**
    1. The fork implements an internal ticker (using std::time::Instant).
    2. When scroll_offset changes (e.g., via a hotkey), instead of jumping instantly, the target offset is set.
    3. On every tick, current_offset interpolates towards target_offset.
    4. **Critical Optimization:** Calls to SetWindowPos for all moving windows must be

batched using BeginDeferWindowPos / EndDeferWindowPos.[13] Without this, the windows will "shudder" independently as the OS processes each message sequentially.

6.4 Input Handling: The Autohotkey Bridge

Since Komorebi lacks native touchpad gesture support, the fork should leverage the existing IPC interface.

- **Mechanism:** A user runs an AutoHotKey script that hooks global touchpad gestures (using specialized libraries like AutoHotInterception).
- **Communication:** When a "3-finger drag" is detected, the script sends a named pipe message to Komorebi: komorebic scroll-pixel-delta <dx>.
- **Response:** Komorebi receives this delta, updates the target_offset, and triggers a layout refresh. This approximates the 1:1 tracking of Niri, limited only by the latency of the IPC and the DWM's SetWindowPos throughput.

---

7. Alternative Approaches

7.1 GlazeWM

GlazeWM is the primary competitor to Komorebi on Windows. Like Komorebi, it is moving toward a Rust codebase.[29]

- **Comparison:** GlazeWM emphasizes simplicity and configuration via YAML, whereas Komorebi emphasizes granular control and CLI scriptability.
- **Feasibility:** Modifying GlazeWM is equally feasible to modifying Komorebi. However, Komorebi's codebase is currently more mature in terms of advanced layout state management. GlazeWM users have actively requested Niri-like features [5], suggesting a fork of either would find a receptive audience. The decision comes down to the developer's preference for the internal architecture of komorebi-core vs. GlazeWM's core.

7.2 The Virtual Monitor Driver (IddSampleDriver) Solution

For a developer seeking *perfection*—absolute tear-free smoothness identical to Niri—the user-space window manager approach (Komorebi/GlazeWM) has an undefined ceiling due to DWM overhead. The "Nuclear Option" is to write an **Indirect Display Driver (IDD)**.

- **Concept:** The driver creates a virtual monitor that is, for example, 10,000 pixels wide. Windows thinks this is a real physical monitor.
- **Niri Simulation:** Komorebi places windows onto this massive virtual canvas.
- **Viewport:** A lightweight full-screen application runs on the *physical* monitor. It acts as a "camera," capturing a 1920x1080 slice of the virtual 10,000px monitor and rendering it.

- **Advantages:** DWM renders the entire virtual strip natively. Scrolling is just texture panning in the viewer app (0% lag, 100% smooth).
- **Disadvantages:** Requires driver signing, high complexity, and potential compatibility issues with anti-cheat software or protected content (HDCP).

---

8. Technical Implementation Roadmap (The "Komorebi-Niri Hybrid")

Based on this deep research, the optimal path for a developer or team is to fork Komorebi. The following roadmap integrates the identified requirements and missing details.

Phase	Objective	Technical Steps
1	**Logic Extraction**	1. Analyze Niri's src/layout.rs.[19] 2. Port the "Infinite Strip" algorithm to a standalone Rust struct (independent of Smithay/Wayland). 3. Ensure logic handles "Focus Centering" and "Gap" calculations.
2	**Core Extension**	1. Modify komorebi-core Workspace struct to support virtual_width and scroll_offset. 2. Replace the existing Scrolling layout logic with the ported Niri logic. 3. Implement the "List" behavior: Window[i].x is strictly dependent on Window[i-1].x + Width.

3	**Win32 Integration**	1. In komorebi (binary), implement the "Buffer Zone" logic. 2. Integrate DwmSetWindowAttribute (Cloaking) for windows outside the buffer zone. 3. Implement BeginDeferWindowPos batching for layout updates.
4	**Input & Animation**	1. Add scroll-pixel-delta command to the CLI. 2. Implement a specialized animation loop for scrolling (distinct from the existing window open/close animations). 3. Create a reference AHK script for touchpad integration.

9. Conclusion

The demand for "infinite scrolling" on Windows represents a desire to break free from the constraints of physical monitor dimensions, a capability that Wayland compositors like Niri have successfully democratized on Linux. While the Windows DWM architecture prohibits a direct port of Niri due to the lack of user-space compositing authority, the **Komorebi fork strategy** offers a highly viable alternative.

By treating Niri's codebase as a reference for layout algorithms and grafting that logic onto Komorebi's mature Win32 event handling, developers can achieve a high-fidelity approximation of the infinite scrolling workflow. The technical challenges—specifically off-screen occlusion culling and animation smoothness—are significant but solvable through the strategic use of DWM Cloaking and DeferWindowPos batching. This approach minimizes engineering effort by reusing the robust foundation of Komorebi while delivering the spatial paradigm shift users are requesting.

Works cited

1. The Future is Niri - a tiling window manager with infinite horizontal scroll : r/rust - Reddit, accessed February 3, 2026, https://www.reddit.com/r/rust/comments/1j9v4gl/the_future_is_niri_a_tiling_window_manager_with/
2. YaLTeR/niri: A scrollable-tiling Wayland compositor. - GitHub, accessed February 3, 2026, https://github.com/YaLTeR/niri
3. Niri – A scrollable-tiling Wayland compositor | Hacker News, accessed February 3, 2026, https://news.ycombinator.com/item?id=45461500
4. Anyone knows a scrolling window manager for windows or enjoys using them? - Reddit, accessed February 3, 2026, https://www.reddit.com/r/windows/comments/1liuoww/anyone_knows_a_scrolling_window_manager_for/
5. GlazeWM vs. Komorebi: Choosing Your Ideal Tiling Window Manager - Oreate AI Blog, accessed February 3, 2026, https://www.oreateai.com/blog/glazewm-vs-komorebi-choosing-your-ideal-tiling-window-manager/94fc792bfe730ef49d7bb52bdb4c2682
6. Tools to achieve a 10x developer workflow on Windows - DEV Community, accessed February 3, 2026, https://dev.to/andresestrella/tools-to-achieve-a-10x-developer-workflow-on-windows-4pk9
7. Desktop Window Manager - Wikipedia, accessed February 3, 2026, https://en.wikipedia.org/wiki/Desktop_Window_Manager
8. What happens with the off-screen front buffers in Windows 7 when DWM is disabled?, accessed February 3, 2026, https://superuser.com/questions/316738/what-happens-with-the-off-screen-front-buffers-in-windows-7-when-dwm-is-disabled
9. SetWindowPos outside screen - Stack Overflow, accessed February 3, 2026, https://stackoverflow.com/questions/15013703/setwindowpos-outside-screen
10. SetWindowPos function (winuser.h) - Win32 apps | Microsoft Learn, accessed February 3, 2026, https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-setwindowpos
11. Development: Design Principles · YaLTeR/niri Wiki - GitHub, accessed February 3, 2026, https://github.com/YaLTeR/niri/wiki/Development:-Design-Principles
12. How to set the position of a Window? - Appeon Community, accessed February 3, 2026, https://community.appeon.com/qna/q-a/how-to-set-the-position-of-a-window
13. SetWindowPos and multiple monitors with different resolutions - Stack Overflow, accessed February 3, 2026, https://stackoverflow.com/questions/22874211/setwindowpos-and-multiple-monitors-with-different-resolutions
14. Excessive Windows DWM usage when screen is locked and display enters standby, accessed February 3, 2026,

https://bugzilla.mozilla.org/show_bug.cgi?id=1924932
15. Finally figured out how to disable windows 10 DWM forced vsync : r/buildapc - Reddit, accessed February 3, 2026, https://www.reddit.com/r/buildapc/comments/iicxhq/finally_figured_out_how_to_disable_windows_10_dwm/
16. window-hiding-behaviour - Komorebi, accessed February 3, 2026, https://lgug2z.github.io/komorebi/cli/window-hiding-behaviour.html
17. niri-ipc - crates.io: Rust Package Registry, accessed February 3, 2026, https://crates.io/crates/niri-ipc
18. A tour of the niri scrolling-tiling Wayland compositor - LWN.net, accessed February 3, 2026, https://lwn.net/Articles/1025866/
19. layout_engine - crates.io: Rust Package Registry, accessed February 3, 2026, https://crates.io/crates/layout_engine
20. Fractional Layout - niri, accessed February 3, 2026, https://yalter.github.io/niri/Development%3A-Fractional-Layout.html
21. Layout - niri, accessed February 3, 2026, https://yalter.github.io/niri/Configuration%3A-Layout.html
22. accessed January 1, 1970, https://github.com/LGUG2Z/komorebi/blob/master/komorebi/src/window_manager.rs
23. accessed January 1, 1970, https://github.com/LGUG2Z/komorebi/blob/master/komorebi-core/src/layout.rs
24. scrolling-layout-columns - Komorebi, accessed February 3, 2026, https://lgug2z.github.io/komorebi/cli/scrolling-layout-columns.html
25. The komorebi.bar.json configuration file reference for v0.1.40 - lgug2z, accessed February 3, 2026, https://komorebi-bar.lgug2z.com/schema
26. [FEAT]: Detailed configurations on layouts · Issue #916 · LGUG2Z/komorebi - GitHub, accessed February 3, 2026, https://github.com/LGUG2Z/komorebi/issues/916
27. [BUG]: focus left/right scrolls focus, but not the window view in scrolling layout (1 column) · Issue #1580 · LGUG2Z/komorebi - GitHub, accessed February 3, 2026, https://github.com/LGUG2Z/komorebi/issues/1580
28. Installation - Komorebi, accessed February 3, 2026, https://lgug2z.github.io/komorebi/installation.html
29. [Media] 12k lines later, GlazeWM, the tiling WM for Windows, is now written in Rust - Reddit, accessed February 3, 2026, https://www.reddit.com/r/rust/comments/1emhbj7/media_12k_lines_later_glazewm_the_tiling_wm_for/
30. [Feature Request] Scrollable tiling like Niri or PaperWM · Issue #1039 · glzr-io/glazewm, accessed February 3, 2026, https://github.com/glzr-io/glazewm/issues/1039