ChatGPT

# Time Tracking System: Requirements Review, Tech Stack, and Development Timeline

The goal is to develop a **cross-platform time tracking application** that runs on Windows, Linux (Wayland), and macOS. This app will provide a minimal always-on-top window for time tracking and also integrate with status bars (like Waybar on Hyprland) for tiling window manager users. Below we review the requirements and propose a suitable technology stack (with a C# preference) and a step-by-step development timeline for a single developer.

## Requirements Overview

- **Cross-Platform UI:** A desktop application with a **small, always-on-top main window** (no standard OS title bar or menu bar, to save space) on Windows, macOS, and Linux. The window should show the current project, a play/pause button for tracking, and the elapsed time. It should be borderless (no OS chrome) and resizable/minimal. In tiling window manager environments (e.g. Hyprland with Waybar), the app should **integrate with the status bar** instead of a separate window (e.g. as a Waybar plugin module).
- **Project Selection & Management:** Users can select the current project (likely via a dropdown or list in the main UI). A separate **Project Settings UI** will allow adding new projects, removing projects, and toggling projects as active/inactive (so that only active projects appear for time logging). This ensures old or irrelevant projects can be disabled and not clutter the selection list.
- **Time Tracking Controls:** A **Play/Pause (Start/Stop) button** to begin or pause tracking time against the selected project. The UI should display the timer for the current session (and possibly total time spent on that project today or in total). When paused/stopped, the session's time should be recorded.
- **Custom Rounding Rules:** The system needs to support configurable rounding rules for time entries. Options should include: rounding recorded durations to the nearest *n* minutes (e.g. 1, 5, 10, 15 minute increments), discarding short sessions under a certain threshold (*n* minutes), and snapping session start/end times to the nearest specified minute mark. For example, if 15-minute rounding is chosen, a session from 10:02 to 10:10 might round to start at 10:00 and end at 10:15, or a 2-minute session might be discarded if the minimum is 5 minutes. These rules should be configurable (possibly globally or per project) in the app's settings.
- **Local Data Storage:** Time tracking data (projects, time entries) should be stored **locally in a SQLite database** on the user's machine. This ensures offline access and privacy. The schema might include tables for Projects and TimeEntries (with fields like project, start_time, end_time, duration, etc.). In the future, the app may sync or export data to an external API, so the design should keep that in mind (e.g. an exporter or synchronization module can be added later).
- **Waybar/Status Bar Integration:** For Linux users on Hyprland (Wayland) using Waybar (or similar status bars in the future), the time tracking info should be viewable in the bar. Ideally, the **same application codebase** can provide this integration – for example, by running as a Waybar custom script module or system tray component – rather than maintaining a completely separate project.

The Waybar integration should display the current project and timer, and potentially allow basic control (like clicking the bar to play/pause the timer).
- **Consistency Across Platforms:** All platforms (Win/Mac/Linux) should use **one codebase** and share as much logic as possible. The UI/UX should be consistent, aside from platform-specific conventions. The main timer window should have no standard title bar (as noted) and should support an always-on-top mode so it can float above other windows when needed. Also, no "file menu" is needed on the timer window, to keep it lightweight. Configuration and project management could be done either in a separate dialog/window or a popover from the main window.

## Proposed Tech Stack

Given the preference for C#, the following tech stack is recommended to fulfill the above requirements:

- **Programming Language & Framework: C# with .NET (Core)** – Using the latest .NET (e.g. .NET 7 or .NET 8) ensures cross-platform compatibility for the core logic. C# is preferred, but this stack also allows using F# or VB.NET if needed. .NET executables can run on Windows, Linux, and macOS with the appropriate runtime or self-contained deployment.

- **Cross-Platform UI Library: Avalonia UI** – Avalonia is an open-source .NET UI framework that is specifically designed for cross-platform desktop applications. It supports Windows, Linux, and macOS natively from a single codebase [1]. The Avalonia framework uses XAML (similar to WPF) allowing us to design a single UI that adapts to each OS. Critically, Avalonia will let us create a custom window without the default chrome (title bar), and we can set it to always be on top. In Avalonia, a **borderless window** can be achieved by setting properties like `ExtendClientAreaToDecorationsHint="True"` and `ExtendClientAreaChromeHints="NoChrome"`, which removes the standard OS title bar and allows a custom-drawn or minimal window [2]. This fits the requirement of having a tiny floating window with only the necessary controls. Avalonia also supports **Wayland** (either through XWayland or in upcoming versions through native Wayland support), which is important for running on Hyprland. In summary, Avalonia allows one to write a **single UI codebase for Windows, macOS, and Linux** with high fidelity to native look and feel, which aligns perfectly with the “same source for all platforms” goal. (Alternatives considered: **UNO Platform** or **Electron** or **Qt**. UNO was less popular and mainly oriented around UWP/Xamarin tech; Electron or web-based was considered too heavy for a small always-on-top utility. **.NET MAUI** was also considered, but MAUI is focused on mobile and has limited desktop capabilities – notably it lacks official Linux support [3], making Avalonia a better choice for a Linux desktop environment.)

- **Database: SQLite (local file)** – SQLite is a lightweight, file-based SQL database which is ideal for local storage of app data. It requires no separate server and works on all platforms. In C#, SQLite can be accessed via **Microsoft.Data.Sqlite** or **System.Data.SQLite**. We recommend using Microsoft.Data.Sqlite (the modern ADO.NET provider) as it is designed with cross-platform usage in mind [4] and integrates well if we later use an ORM like Entity Framework Core. For a simpler approach, one could also use a micro-ORM like **Dapper** or an active record pattern via **SQLite-net** library [5], but given the project scope, using raw SQL or EF Core with the SQLite provider are both viable. The database will contain a Projects table (with fields like ProjectID, Name, Active flag, etc.) and a TimeEntries table (with fields for entry ID, project ID (foreign key), start time, end time, duration, and possibly a note or flags for rounding). We will ensure all database code is cross-

platform (which it will be under .NET) and that the app handles the database file path appropriately on each OS (e.g. storing it in user's AppData or ~/.local/share directory).

- **Time Tracking Logic:** The core logic will be implemented in C# (as part of the application code or a separate class library). This includes managing the timer (likely using `System.Timers` or a dispatcher timer to update the UI every second), handling play/pause, and applying the **rounding rules** when saving an entry. The rounding configuration can be stored in the database or a config file, and the logic will adjust the start/end times or durations according to the user's selected rounding option (nearest N minutes, snap start/end to N-minute marks, and drop short sessions below N minutes). These computations are straightforward to implement with time arithmetic, and they will be tested for accuracy.

- **Waybar (Status Bar) Integration:** Instead of writing a completely separate plugin in C++, we can leverage Waybar's support for custom script modules. **Waybar allows executing an external script/program and displaying its output in the bar** [6]. We will take advantage of this by providing a **CLI mode or a small companion script** for our time tracker:

- One approach is to have the main application executable support a command-line argument (for example, `timetracker --status`) that, when run, outputs the current active project and elapsed time (and possibly icon or status) to stdout, then exits. The Waybar config can call this periodically (e.g. every few seconds) to update the bar text. Waybar's custom module will capture that output and show it in the bar.
- Additionally, Waybar can be configured to send signals or run commands on click [7]. We can use this to our advantage by specifying that a click on the time tracker module triggers our app to toggle the timer (for instance, by running `timetracker --toggle` which signals the main app to start/stop). Implementation-wise, this could be done by the CLI mode writing a small flag or using IPC (inter-process communication) to talk to the running app. If implementing IPC is complex, a simpler workaround is to have the main app also continuously update a status file or database entry that the script reads, or to allow the CLI `--toggle` to directly manipulate the database (e.g. create a new TimeEntry or end the current one) if the main app isn't running. These details can be refined during development.
- By using the same codebase for this (e.g. the core logic library), we ensure no duplicate logic. The script/CLI and the GUI will both use the same underlying timer data (possibly interacting through the DB).

- This approach will also generalize to other status bars (for future needs). Many Linux status bars (polybar, i3blocks, etc.) also allow scripts for custom modules. On Windows or macOS, a similar integration could be a system tray icon showing the timer (Avalonia might allow creating a tray icon via platform-specific features or using a library). That could be a future extension, but not critical for the first version.

- **Development Frameworks & Tools:** We'll use standard development tools like Visual Studio or VS Code with the .NET SDK. For Avalonia, the Avalonia VS Extension or AvaloniaUI previewer can help design the XAML UI. We will target .NET Core so the app can be published self-contained for each OS (meaning the user doesn't need to install .NET separately). Version control (git) will be used to manage the source, and we might set up CI pipelines to build for Windows, Linux, Mac if desired (this can be considered in later stages, not immediately required).

In summary, **C#/.NET with Avalonia** covers the cross-platform GUI needs and aligns with the preference for C# [1]. **SQLite** meets the local storage requirement. This stack is modern, single-developer friendly, and has a thriving community and documentation to help along the way. Next, we outline a development timeline to implement this system.

## Development Timeline

Below is a proposed timeline broken into phases/weeks for a single developer building this time tracking application. The schedule can be adjusted based on actual progress, but this provides a framework:

1. **Week 1-2: Planning & Setup**
2. **Requirements Refinement:** Revisit and refine the requirements in detail. Sketch out the user interface layout for the main timer window (project selector, timer display, play/pause button) and the project management UI. Define how rounding options will be presented to the user (perhaps in a settings dialog or as part of project settings).
3. **Tech Stack Setup:** Initialize the project using .NET (e.g., create a solution with an Avalonia app project). Ensure the development environment is ready on at least one platform (Windows or Linux) and that Avalonia is properly installed. Create a basic "Hello World" window in Avalonia to verify that the project runs on Windows, Linux (Hyprland/Wayland or X11), and macOS. This might involve testing an Avalonia app on each platform (using XWayland if necessary for initial tests).
4. **Architecture Design:** Plan the architecture – create a project structure with perhaps three components:
    - Core library (for models like Project, TimeEntry, and business logic such as time calculation and database access),
    - UI project (Avalonia) that references the core,
    - Optionally, a CLI or integration module for Waybar (this could be just part of the core or a separate console app that uses the core).
      Decide on the data model for the database and outline the SQLite schema (e.g. writing SQL DDL for tables or planning to use EF Core migrations).

5. **Choose Libraries:** Add NuGet packages for SQLite access (e.g. Microsoft.Data.Sqlite or an ORM). Also include any utility libraries (for example, if using reactive UI patterns or MVVM in Avalonia, bring in those packages). Confirm that these libraries are cross-platform compatible.

6. **Week 3-4: Core Functionality Development**

7. **Implement Database Layer:** Create the SQLite database file and connection setup. Using a path in user's local app data (on Windows) or home directory (on Linux/macOS). Implement code for initializing the database (creating tables if not exist). This could be done via simple SQL execution or using Entity Framework Core Code-First migrations if that route was chosen. Test that you can read/write to the database from each platform (for example, ensure file paths and permissions are handled correctly).
8. **Project Management Backend:** Implement functions in the core library to add a project, remove (or mark inactive) a project, and retrieve active projects. These functions will interface with the SQLite DB. Also include validation (e.g. no duplicate project names, etc.).
9. **Time Tracking Logic:** Implement the core timer logic. Likely, create a `TimeTracker` service/class in the core that can Start, Pause, and Stop timing. When Start is called with a selected project, record

a start timestamp (and possibly create a new TimeEntry record in the DB with start time). On Pause/Stop, calculate the elapsed time, apply the rounding rules (if configured), update the TimeEntry in the DB with the final end time and rounded duration. If a session is below the minimum threshold (e.g. <5 minutes and user set 5-minute minimum), decide whether to discard it (perhaps mark it as discarded or not save it at all).

10. **Rounding Rules Config:** Define a data structure or class to hold the rounding settings (e.g. an enum or constant for rounding increment, a boolean or value for minimum threshold, and a flag for snapping start times). Implement utility functions to perform rounding calculations (for example, a function that given a start & end DateTime and a rounding configuration, returns adjusted start/end). Write unit tests (if possible) or simple console tests for these rounding functions to ensure they work as expected (e.g. test that 2 minutes rounds to 0 if threshold is 5, test that 7 minutes rounds to 5 or 10 appropriately, etc.).
11. **Basic CLI Mode:** Implement a very simple console output as a proof of concept – for example, if the app is run with `--status` argument, print "No active project" or "Active: Project X – 0h 5m" to ensure the core can output status. This sets the stage for Waybar integration later.

12. **Preliminary Testing:** By end of week 4, you should have the core logic working in a rudimentary way. You might write a short console app or use the immediate window to simulate starting and stopping a timer, and verify the entries get written to the database with correct rounding. This core foundation will make the next phase (UI) easier.

13. **Week 5-6: User Interface Development**

14. **Main Window UI:** Using Avalonia XAML, design the main timer window. This likely involves a ComboBox (or dropdown) to choose a project, a label or text block to show the elapsed time, and a Button or ToggleButton for Play/Pause. Since there's no standard window chrome, also implement a way to move the window (e.g. Avalonia allows making a borderless window draggable if you handle events – perhaps allow dragging by an empty area or a special grip element). Also, set the window properties to `Topmost = true` (always on top) and remove the minimize/maximize buttons. Ensure the window starts at a small size (maybe just enough to fit the controls).
15. **Data Binding & MVVM:** To keep logic separate from UI, consider using Avalonia's MVVM support. Create view models for the MainWindow and for Project list, etc. Bind the project dropdown to the list of active projects (populated from the database via the core library). Bind the play/pause button to commands that call the TimeTracker service. Bind the timer display to a property that updates every second (you might use a Timer to tick and update, or leverage Avalonia's `DispatcherTimer`). This way, the UI will automatically reflect the current timing state.
16. **Project Management UI:** Create a secondary window or dialog for managing projects. This could be a simple window with a ListBox of all projects with checkboxes or toggles for active/inactive, and add/remove buttons. Hook up these controls to the core library functions (add project, deactivate project, etc.). Make sure to update the main window's project list after changes. This UI should also be borderless or at least minimal, but since it's an occasional-use window, it could have a title bar if needed for dragging (optional).
17. **Settings for Rounding:** Provide a UI to adjust rounding rules. This might be a section in the project management dialog or a separate settings dialog. Options could be dropdowns or numeric inputs for "round to nearest X minutes" and "minimum session length" and toggles for snapping start/end. Store these settings in the database (could be a simple table for settings or even stored in the Projects table if each project can have its own rounding setting). For the first version, a global setting

is simpler. Ensure that changes to these settings will affect how new sessions are recorded (document that existing entries won't retroactively change).

18. **Polish UI Elements:** Add visual indicators if needed, such as an icon on the play/pause button (▶ / ⏸ symbol) that toggles, maybe color changes when timing is active vs paused. Make sure text is readable on all platforms (Avalonia default styles should handle most of this).

19. **Testing on Each Platform:** At this stage, run the UI on Windows, Linux, and macOS to verify it appears correctly. On Windows and macOS, the window should be borderless and on top. On Linux/Hyprland, test in Wayland: since Avalonia might run via XWayland, check that the window is indeed floating. If Hyprland tiling tries to tile it, you might need to set a window property or Hyprland rule to float it (some Wayland compositors allow matching window classes to float). Verify the basic functionality: selecting a project, starting timer, stopping, and that the DB is updated. This may reveal some platform-specific tweaks (e.g. file paths differences, or needed libraries for SQLite on Linux – ensure `libsqlite3` is present, etc., usually Microsoft.Data.Sqlite bundles SQLite native or it must be installed, so double-check that).

20. **Week 7-8: Integration & Advanced Features**

21. **Waybar Integration Development:** Now that the core app is functional, work on the Waybar module integration. Create a script or use the application's CLI mode to output status. For example, you might write a small bash or Python script that calls `timetracker --status` and parses output, but a cleaner method is to have `timetracker --status --format=json` to output JSON like `{"text": "ProjA 0h10m", "tooltip": "Project A – 10 minutes tracked", "class": "tracking"}` which Waybar can ingest directly [8]. Implement this in the app's code. Test running this from a terminal to see that it prints the correct status.
22. **Waybar Config Setup:** Configure Waybar (on your Hyprland environment) by editing the Waybar config file to add a custom module. For instance, in the Waybar config JSON:

    ```
    "custom/timetracker": {
        "format": "{} {text}",
        "exec": "/path/to/timetracker --status",
        "interval": 5,
        "on-click": "/path/to/timetracker --toggle"
    }
    ```

    This would call the `--status` every 5 seconds to update, and call `--toggle` when clicked. Adjust the format as needed (maybe include an icon or prefix). Document these instructions for end users as well.
23. **Implement Toggle/Control via CLI:** Ensure the `--toggle` command works. This likely means if the app is running, it should communicate to the running instance to start/stop. This can be done via a simple file lock or a network socket or by writing an "intention" to the database that the main app polls. As a simple approach, if the main app is always running while tracking, we could skip complex IPC by requiring the user to open the app. If that's acceptable, `--toggle` could directly manipulate the DB: if no active running entry, insert a new one; if there is one active (perhaps mark active in DB), then update it to stopped. This could allow starting/stopping without the main UI, which might

actually be fine. Just be careful with concurrency (two toggles quickly, etc.). Test that this works from a click in Waybar (you might simulate by running the command manually).

24. **Cross-Platform Tray (Optional):** As an extra, on Windows/macOS you might consider adding a system tray icon with a context menu to start/stop and show the window. Avalonia doesn't have built-in tray support in all platforms, but there are plugins or one can use platform-specific interop. This could be scheduled later or skipped initially. It's not core to the requirements, but good to note for future.
25. **Data Review and Export:** Implement a simple view or export if needed – for example, a way to list all time entries or to summarize time per project. This wasn't explicitly requested, but a minimal report or log view could be helpful to verify the app's tracking. At least ensure the data can be opened with external tools if needed. Possibly provide a CSV export of tracked time for a given date range (this can be a stretch goal if time permits).

26. **Apply Refinements:** With integration in place, refine any rough edges discovered. This includes making sure the rounding rules are correctly applied in all cases, UI doesn't freeze (use async DB calls or background tasks for any heavier operations), and the app handles edge cases (like switching project mid-timer – perhaps disallow that until stopped, or auto-stop and start new). Also handle if the app is closed while a timer is running – maybe on next launch detect an "active" entry and offer to continue or end it. These details improve user experience.

27. **Week 9-10: Testing, Polish, and Release Prep**

28. **Comprehensive Testing:** Now perform thorough testing on all target platforms:
    - **Windows:** Does the app run without needing admin? Is the always-on-top behavior working? (On Windows, a Topmost window should float above others; verify this.) Does the borderless window look OK (no weird transparent artifacts)? Test installing/running on a clean Windows machine or VM to ensure the SQLite dependency is fine.
    - **Linux (Wayland/Hyprland):** Test the Waybar module end-to-end: the bar shows the project/time, clicking toggles the timer, etc. Also test on a different environment if possible (e.g. GNOME or KDE Wayland session) to ensure the app at least functions (it might appear as XWayland if Avalonia's Wayland support isn't fully ready). Check that closing the main window doesn't quit the app unexpectedly if you intend it to run in background for Waybar (you might decide the app runs in background if closed while tracking; implement that if needed via a tray or just have an option to minimize to tray).
    - **macOS:** Ensure the app feels native enough – macOS might treat an always-on-top window differently (make sure it appears above other apps when active). Also, Mac usually expects a menu bar for apps; Avalonia can run without one, but be mindful that closing windows on Mac doesn't always quit the app. Possibly add a simple menu with "Quit" for Mac users or ensure Command+Q works.
    - **Edge Cases:** Test rounding rules thoroughly: try different rounding increments and thresholds, ensure the math is correct (no off-by-one-minute issues). Test adding/disabling projects and see that the UI updates accordingly. Test multiple sessions in a row and ensure data is logged correctly (e.g. start Project A for 5 min, stop, then start Project B, etc.).
29. **Performance & Optimization:** The app is small, but ensure that the timer update (running every second) isn't consuming excessive CPU. Avalonia is efficient, and a one-second update should be fine. The SQLite writes for each session end are quick, but if you want, you could batch or defer writes. Typically not needed for a time tracker since writes are infrequent (only on stop). If using

Entity Framework, ensure it isn't tracking too many objects (maybe use NoTracking queries or dispose context after each operation to avoid memory bloat).

30. **UI/UX Polish:** Tweak any visual issues identified during testing. For example, if text doesn't fit in the window at smaller sizes, adjust the layout or minimum window width. Perhaps add tooltips or labels to make it clear what each control does (e.g. hover text "Play/Pause Timer"). Ensure keyboard accessibility (maybe allow starting/stopping with space bar or a global hotkey if that's feasible – though global hotkeys on Wayland are tricky without compositor support, so this might be skipped or only implemented on Windows/macOS if desired).
31. **Documentation:** Create a brief README or Help section within the app that explains how to use it, how the rounding rules work, and how to configure the Waybar (or other bar) integration. This is important especially for the first users of your app (which might just be you, but documentation will help if you release it or come back to it after a break).
32. **Packaging & Deployment:** Set up the project for publishing on each platform. For Windows, you might create an installer (MSIX or use a self-contained EXE). For macOS, create a .app bundle (Avalonia can produce one, or you use `.dotnet publish` with `-r osx.ARM64` or x64 as needed and then package). For Linux, consider distributing as an AppImage or just a zip with the binary, or even a Flatpak in the future. At least ensure you can publish a self-contained Linux build that includes all needed libraries (including the .NET runtime). Also ensure SQLite native library is present in the output (Microsoft.Data.Sqlite usually handles this via dependency on `e_sqlite3` for each platform).

33. **Buffer and Final Adjustments:** If any tasks slip from earlier weeks, use this time to catch up. By the end of week 10 (about 2 to 2.5 months in), you should have a **fully functional MVP (Minimum Viable Product)**: the time tracker app running on all platforms, with local data storage, basic project management, rounding logic, and Waybar integration.

34. **Week 11+ (Post-MVP Enhancements – *optional*)**
    *(This goes beyond the initial timeline, but notes future improvements if you choose to continue development after the initial release.)*

35. **Synchronization/API Integration:** Begin exploring how to send data to an external API or server if needed (as mentioned in requirements for possible extensions). This could involve designing an export function or integrating with a time-tracking service's API. Likely, you'd implement this as an optional sync feature that sends new time entries to a server. Ensure this is done securely (maybe use an HTTP client with authentication tokens, etc.).
36. **Enhanced Reporting:** Add views for past tracked time, like a daily/weekly summary per project, or the ability to edit past entries (in case of mistakes).
37. **Improved Waybar/Bar Integration:** If Waybar supports it, possibly refine the module to show more info or controls (Waybar can show tooltip on hover, so maybe list last few entries in tooltip). For other bars like Polybar, create equivalent scripts. On Windows, consider adding a small overlay or widget.
38. **User Preferences:** Expand settings to include things like launching on startup, minimizing to tray, auto-updates, etc., depending on user feedback.
39. **Testing & Feedback:** If releasing to others, gather user feedback and iterate.

**Deployment of the MVP should be achievable by around Week 10**, given consistent progress. As a single developer, this timeline is ambitious but manageable, since the project scope is moderate. The use of high-level frameworks (Avalonia and .NET) avoids the need to write platform-specific GUI code and lets you focus

on features. Each phase above builds on the previous, and by following this plan, you will gradually implement all required features in a maintainable way.

## Conclusion

By leveraging **C# and Avalonia**, we satisfy the cross-platform and UI requirements with one shared codebase [1]. The local data storage is handled through **SQLite**, which is lightweight and cross-platform friendly [4]. The application's design (modular core logic with a separate UI layer) ensures that features like Waybar integration can reuse the same underlying data and functions, fulfilling the "same source for all interfaces" goal. The proposed development timeline (approximately 2-3 months for an initial version) breaks the work into manageable chunks, from initial setup, through core logic, UI implementation, integration, to final testing and packaging. Following this plan will result in a **small, efficient time tracking app** that meets the requirements: always-on-top minimal window for desktops and seamless status-bar integration for tiling window environments, with robust time tracking features including project management and custom rounding rules. Good luck with your development – with careful planning and iterative progress, you'll have your cross-platform time tracker up and running in no time!

**Sources:**

- Avalonia cross-platform GUI support – works on Windows, macOS, and Linux [1].
- Avalonia borderless window (no title bar) achievable via window properties [2].
- Microsoft.Data.Sqlite – a cross-platform SQLite library for .NET [4].
- Waybar custom module can display output of an external script (for integration) [6].

---


[1] [3] Is Avalonia the best solution for cross platform desktop apps? : r/learncsharp
https://www.reddit.com/r/learncsharp/comments/12qovcq/is_avalonia_the_best_solution_for_cross_platform/

[2] avalonia - AvaloniaUI - how to change the 'style' of the window (borderless, toolbox, etc) - Stack Overflow
https://stackoverflow.com/questions/65748375/avaloniaui-how-to-change-the-style-of-the-window-borderless-toolbox-etc

[4] .NET 8 WinForms Migration – System.Data.SQLite vs Microsoft.Data.Sqlite - Microsoft Q&A
https://learn.microsoft.com/en-us/answers/questions/2284458/net-8-winforms-migration-system-data-sqlite-vs-mic

[5] c# - Cross-platform Sqlite - Stack Overflow
https://stackoverflow.com/questions/13016578/cross-platform-sqlite

[6] [7] [8] waybar-custom(5) — Arch manual pages
https://man.archlinux.org/man/extra/waybar/waybar-custom.5.en