ChatGPT

Product Requirements Document: Offline Timeline Activity Tracker (macOS)

Target Platform

- **Platform:** macOS (Apple Silicon M1/M2/M3 series) – The application will run natively on Apple Silicon MacBooks. It should be built with **Swift 5+** and **SwiftUI** for a native look and optimal performance on Apple Silicon.
- **OS Compatibility:** Target macOS 12.0+ (Monterey or later) to leverage modern SwiftUI features (including `MenuBarExtra` for menu bar apps). The app should be optimized for ARM64 architecture and take full advantage of Apple Neural Engine if any on-device ML is later needed (though none is currently planned). Intel Macs are not a primary target (the app may run under Rosetta if needed, but native Apple Silicon support is required).
- **Frameworks & Libraries:** Use native macOS frameworks:
- SwiftUI for UI components and layouts.
- Combine or Grand Central Dispatch for background tasks (continuous logging).
- **AppKit** integration for any needed low-level functionality not provided by SwiftUI (e.g. menu bar item via `NSStatusBar` if needed, or handling system permissions).
- **SQLite** (via an embedded library or Swift's `sqlite3` C API, or a Swift package like SQLite.swift) for local data storage.
- **Accessibility API / CoreGraphics API:** Use Apple's APIs to detect the active application and window title (e.g. `NSWorkspace` notifications for app switches, and Accessibility or CoreGraphics to fetch window titles). The app will require user permissions for these (Accessibility and Screen Recording – see Privacy section).
- **Distribution:** The app will be a standalone macOS application (likely distributed outside Mac App Store due to the need for Accessibility privileges; if Mac App Store distribution is desired, ensure it meets sandbox requirements and uses the proper entitlement for screen recording if available). No server or web components are involved since this is strictly offline.

Overview

This product is an **offline, timeline-based activity tracking application** for macOS. The purpose of the application is to **automatically record a user's activity throughout the day** on their Mac and present it in a coherent timeline. It logs which application is in use, the title of the frontmost window/document, and periods of inactivity (idle time). Users can also insert manual markers (with custom text notes) to annotate what they were doing at specific times (useful for logging offline activities or context). The app will **auto-classify activity segments into categories or tags based on user-defined rules**, enabling users to see how their time is spent (e.g. "Development", "Communication", "Entertainment" etc., as defined by the user). All tracking and classification happens **entirely offline** on the user's machine – **no internet access or cloud sync** is involved, ensuring privacy.

The user interface will be **clean, minimalistic, and elegant**, providing intuitive timeline views and summary dashboards. Users can review their activity by day in detail (chronological timeline with exact apps and tasks) or in aggregate over longer periods (daily/weekly/monthly/yearly summaries). The UI will primarily be accessible from the **menu bar** as a small utility (with a menu bar icon), reflecting a lightweight, always-on design. Despite running continuously in the background to log data, the app will be optimized for efficiency and have a premium feel in terms of design and responsiveness.

In summary, this application acts as a personal “chronicle” of the user’s computer usage, helping them understand where their time goes each day without compromising privacy. All data is stored locally in a SQLite database; the user’s activity data never leaves their machine. The following sections detail the functional requirements, non-functional criteria, data design, rules engine, UI specifications, and privacy considerations for this product. This document is intended to be unambiguous and detailed enough for developers (or AI like Codex) to implement the application without further clarification.

Core Functional Requirements

The application must fulfill the following core functions and behaviors:

- **Continuous Activity Logging:** The app will continuously monitor and record the active usage of the Mac. This includes capturing the **currently active application name** and the **title of the frontmost window** of that application. Entries in the log should include timestamps for when each activity starts and ends. The logger must also detect **idle periods** when the user is not actively interacting with the computer:
- Use system APIs or polling to determine user idle time (e.g. using `CGEventSourceSecondsSinceLastEventType(..., kCGAnyInputEventType)` to get time since last input event [1]). If no keyboard/mouse input occurs for a specified **idle threshold** (default 5 minutes, configurable), mark the start of an idle period. When user activity resumes, mark the end of that idle period.
- Each distinct period of user activity or inactivity should be recorded as an **activity session**. For example, if the user works in “Xcode – ProjectA.swift” from 9:00 to 9:50, then is idle from 9:50 to 10:00, and then uses “Safari – example.com” from 10:00 to 10:30, these should be three logged sessions (Xcode session, Idle session, Safari session).
- **Accuracy:** The logging should capture changes **in real time**. Whenever the user switches to a different app or the front window title changes (within the same app), the logger should record a new entry. This may involve subscribing to `NSWorkspace.didActivateApplicationNotification` for app switches and using Accessibility APIs (AXUIElement) or CoreGraphics (`CGWindowListCopyWindowInfo`) to detect window title changes. Polling can be used as a fallback (e.g. checking the frontmost window title every 1-5 seconds) to catch changes that aren’t caught by notifications. The goal is to never miss a context switch, but also avoid redundant logs when nothing has changed.

- **Idle handling:** Idle time segments should be recorded with a special designation (e.g. an entry with `app_name = "Idle"` or a boolean flag). Idle tracking uses the threshold (user-adjustable) to decide when to start an idle session. If the user returns from idle, a new active session begins (even if it’s the same app as before idle, the idle break creates a boundary between sessions).

- **Segment Merging (Avoiding Redundant Entries):** The logging system should intelligently merge or avoid creating duplicate consecutive entries for the same continuous activity:

- If the user remains in the **same application and window** without interruption, the app should treat it as one single session rather than creating multiple entries. For example, if a window title doesn’t change and the user continuously works in it, the logger should update the end-time of the existing log entry rather than inserting new ones. This ensures the timeline isn’t cluttered with repeated entries for the same activity.

- If an activity is interrupted by a different app or idle period and then returns to the same app/window later, it should be logged as a separate session (since the continuity was broken). **Do not merge across interruptions.** For instance, if the user used Chrome at 11:00, switched to Mail at 11:10, then back to Chrome at 11:15, there should be two Chrome sessions (11:00–11:10 and 11:15–...) separated by the Mail session.
- The merging logic primarily applies to avoid spamming the log with identical entries in cases where the tracking mechanism polls frequently or the window title has minor updates. Adjacent log events with identical app and window title should be combined into a single entry with an expanded time range. This merging can be handled in real-time (by not closing the previous entry until something changes) or as a post-processing step on the data before display – but the end result must present as continuous blocks of activity.

- **Edge cases:** If a user manually inserts a marker (see below) in the middle of a session or if a short idle period occurs (shorter than the threshold) but we still get an idle event, the implementation may choose to merge those back into the surrounding session or treat them separately. As a default, treat any idle >= threshold as a break (separate segment). Manual markers do not split sessions; they are annotations at a point in time.

- **Manual Marker Insertion (Timestamped Notes):** The user must be able to insert **manual markers** into the timeline, in order to log what they are doing or add notes at specific times (especially useful for non-computer tasks or context like “Meeting from 2-3pm” or “Break/Lunch” etc.). Requirements for markers:

- The app will provide a quick way to add a marker with a text note and an automatic timestamp of the current time. This should be accessible from the menu bar interface (for example, a text field or an “Add Marker” button in the drop-down menu) and via a **global keyboard shortcut**. The keyboard shortcut should be configurable (with a sensible default, e.g. Ctrl+Option+M, unless that conflicts) so that the user can press it to bring up the marker input quickly from any app.
- When adding a marker, the user can enter a brief textual note (e.g. “Lunch break” or “Call with Client X”). The marker will be saved with the exact time it was created (to the second). If the user invokes the marker command while the app is in the background, the app should still record it (possibly via a global hotkey handler).
- The manual marker does **not** necessarily split an existing activity session; it’s an overlay or annotation on the timeline. For example, if the user was idle from 1:00–2:00 and then inserts a marker “Lunch”, that marker at 1:00 might label that idle block as Lunch. However, implementation-wise we will treat it as a separate data entity (does not alter the existing Idle entry). The UI can show the marker alongside the timeline (like a flag or a line at that time).
- Markers should be visible on the timeline view (e.g. as a labeled tick or separate line at the specific time, perhaps with an icon or distinct color). They should also be stored in the database for permanence. Markers allow the user to later recall what an idle period or ambiguous activity was (for example, if they step away from the computer for a meeting, on return they insert a “Meeting with Team” marker at the time they left).

- The app should allow viewing and editing/deleting past markers (perhaps via the timeline or a list in the UI). At minimum, if a user adds a marker in error, they should be able to remove it (maybe by right-clicking the marker on the timeline or via a separate markers list interface).

- **Customizable Rule Engine for Auto-Tagging:** The application will include a rule-based classification system that can **automatically tag or categorize activity segments** based on user-defined rules. This allows the timeline and stats to show meaningful categories (e.g. “Work”, “Browsing”, “Design”, etc.) instead of raw app names only. Key points:

- The user can define a set of **matching rules** that examine each activity session's attributes (app name and/or window title) and assign a **tag** (label/category) if the rule conditions are met. For example, a rule could specify: *If app name is "Slack", tag the session as "Communication"*; or *If window title contains "Project X", tag as "Project X"*.
- Rules should support multiple conditions combined with logical AND (within a rule). For instance, a single rule could require both a certain app and a keyword in the window title to match. (E.g. App = "Google Chrome" AND window title contains "Stack Overflow" could be tagged "Research"). We also want to support some level of OR or pattern matching: for example, the rule format should allow specifying multiple values for a field (like app name is either Xcode or VSCode) or pattern matching (window title contains one of a list of keywords). At minimum, **substring matching** for window titles is required (case-insensitive contains). Regex or wildcard matching can be supported for advanced use but is optional.
- Each rule will have an associated **output tag** (a short text label, e.g. "Work/Development" or "Social Media"). When a session matches a rule, that session is labeled with the rule's tag. Tags can be thought of as categories or keywords.
- **Priority and conflict resolution:** The rules engine must handle situations where multiple rules could match the same activity. Each rule will have an integer **priority** value. A higher priority number means the rule takes precedence over lower priority ones. When classifying a session, the engine should evaluate all rules that apply; if more than one rule matches, the **highest-priority rule wins** and its tag is applied to that session. If two matching rules have equal priority, the tie-break can be decided by the order in the rules list (e.g. later rules override earlier ones, or vice versa – we will define it as *later definitions override earlier if priorities are equal* to give deterministic behavior). Users will be instructed to use priorities to avoid overlaps, but the engine will still handle overlaps predictably.
- **User-Editable Rules:** The rules are fully user-configurable. The rules should be stored in a human-editable format (YAML or JSON) so that advanced users can edit them easily. The application will provide a built-in **Rules Editor** UI (detailed later) to allow users to add, remove, or modify rules without directly editing files if they prefer. Rules can also be imported/exported by copying the YAML/JSON.
- The rules engine should apply in **real-time** for new incoming data. As soon as a session is logged (or while it's ongoing), it should be evaluated against the rules and tagged accordingly (the tag can be assigned once the session starts, or upon its end when full title is known – but assigning as early as possible is fine). The assigned tag for each session should be stored (or derivable) so that the UI can display categorized data.

- Changes to rules should affect *future* logging immediately. The application does not necessarily need to retroactively re-label past logged data on its own, but it **should allow** the user to trigger a re-classification on existing data if they change their rules. For example, after editing rules, the user might click "Reapply Rules to Past Entries" (or this could happen automatically) so that historical entries get updated tags according to the new rules. If implementing automatic retroactive tagging is complex, an acceptable approach is to only use new rules going forward, but **no ambiguity**: if feasible, provide a way to re-categorize historical data as well (maybe on user demand). In any case, since all data is local, the user could delete or reset tags manually if needed – but the preferred approach is automated via the rules engine.

- **Time-Based Statistics and Summaries:** The application will provide aggregated **statistics on activity over various time spans** (day/week/month/year), enabling the user to review their productivity or habits at a glance.

- At a basic level, the app should compute the total time spent per tag/category and/or per application for a given time range. For example, "Today: 3h on Development, 1h on

Communication, 30m idle, 1h on Other". Similarly for the past week, month, and year. These stats help users see trends (e.g., weekly totals) and big-picture allocation of time.
- **Daily:** The daily view is essentially the timeline (detailed in UI section) showing a breakdown of that specific day. In addition to the timeline, a summary of that day (total hours by category or app) could be shown.
- **Weekly:** Provide a summary for the last 7 days (or any selected week). This might include a bar chart of each day's total active time, and breakdown by tag for the whole week. For example, "Week of Oct 1–7: 40h total tracked (X% Work, Y% Meetings, Z% Idle…)".
- **Monthly and Yearly:** Similar concept, but aggregated by month or year. For a month, possibly show each week's totals or an average per day, and the category breakdown for the entire month. For a year, show each month's total or an average per month, plus overall category split for the year.
- The UI will contain a **Stats Dashboard** to present these summaries (detailed below). Calculations should be done locally using the logged data. Ensure that the queries on the SQLite database (for summing durations by category/app etc.) are optimized (use indices where appropriate) so that even a year's data can be aggregated quickly without lag.
- The user should be able to **select the timeframe** for stats easily (e.g. a toggle or dropdown for "Today/This Week/This Month/This Year" and possibly navigation to previous periods). It might default to showing the current week, current month, etc., with options to go back or specify a custom range if needed (custom range is a nice-to-have, not explicitly required).

- **Idle time handling:** Idle periods should be included in summaries (possibly labeled as "Idle" category) so the user sees the full 24h accounted (or they can choose to exclude idle if they only care about active time – possibly a toggle to include/exclude idle time from stats). By default, include idle as its own category in reports.

- **Local Data Storage (SQLite Database):** All captured data (activity logs, markers, rules/tags) must be stored locally on the user's machine in a persistent way. Use a **SQLite database** file to store this information in a structured form:

- The SQLite database ensures fast queryable storage for timeline data and stats. It will reside in the user's file system (for example, under `~/Library/Application Support/<AppName>/activity.sqlite`).
- Every logged activity session, marker, etc., is inserted into this database. The schema is defined in detail in the Data Structures section, but at a high level: there will be tables for **Activities**, **Markers**, **Tags**, and possibly a separate table or structure for **Rules** (though rules might also live in a separate file).
- The database should be updated in real-time as events occur. It should also be robust against crashes (use transactions for inserts/updates so as not to corrupt the DB). Using SQLite's WAL (write-ahead log) mode is recommended for concurrency safety and performance.
- Since data can grow over time (daily logging), the design should consider performance over months/years of data. Indices on timestamp fields or others will be used to keep queries (especially for stats) efficient. All data is stored indefinitely by default (unless user chooses to purge old data manually; no automatic deletion is required by these specs).
- **No cloud sync:** The storage is solely local. There is no requirement to sync data to iCloud or any server. Backups are the user's responsibility (they could copy the DB file if needed). The app should possibly provide an **export** option (like export to CSV or JSON) for the user to manually back up or analyze data externally, but this is a stretch goal (not mandatory).
- **Local Rule Storage:** The user's custom rules and settings are also stored locally (for instance in a configuration file or within the database). Specifically, rules are to be saved in a **YAML or JSON** formatted file (e.g. `rules.yaml` in the app support directory) to facilitate easy editing and

possibly version control by the user. The app will read this file on startup (and watch it for changes or reload when the user edits rules via the app).

- The SQLite DB will not be excessively large since it's mainly text and timestamps; however, after long-term use (say years), the user might accumulate tens of thousands of records. The app should handle this gracefully. We will ensure the schema is normalized (e.g. using tag IDs instead of repeating tag text for each entry) to keep size and redundancy down.

- **Full Offline Operation (No Internet Access):** The application **must operate entirely offline.** It should **never** make any network requests or send any data out:

- All processing (logging, classification, statistics) is done on-device. The app does not require an internet connection for any functionality.
- There will be **no telemetry, no analytics, no usage data sent** to any server. The user's activity data is strictly private to them. This is a core principle of the product due to privacy considerations.
- Because of this requirement, any libraries or frameworks used must also respect offline operation (e.g. avoid anything that might ping an online license server, etc.). The app will not even check for updates automatically – updates would be delivered manually by the user downloading a new version (unless the user has it through App Store which handles updates, but the app itself will not perform update checks).
- The absence of internet usage should be verifiable: developers should ensure no accidental endpoints are called. If possible, the app might expose a "Network Activity: None" message in a privacy info section, reinforcing to the user that it does nothing online.
- **Auto-classification** must be based on the user's rules only (no cloud-based AI). If categorization suggestions or default rules are provided, they should be hard-coded or included locally, not fetched from a server. (For example, we might include a default rule set for common apps, but that would be shipped within the app or configured by the user, not pulled from the internet.)

Non-Functional Requirements

In addition to core features, the product must meet these non-functional criteria to ensure a good user experience and maintain the product's principles:

- **Clean, Minimalist UI Design:** The user interface should be uncluttered, modern, and **premium-looking**. Use native SwiftUI components styled to fit macOS aesthetics (respect system font, color schemes, and spacing per Apple's Human Interface Guidelines). The design should make good use of typography and whitespace so that information is clear at a glance. Prefer simple visualizations (timelines, bars, pie charts) and avoid overly complex or dense tables. The overall feel should be similar to a high-quality Mac utility: minimal chrome, focus on content (the timeline and stats themselves), and subtle use of color and icons. Support both **light and dark modes** seamlessly. Transitions and animations (e.g. expanding a timeline entry, switching tabs) should be subtle and smooth to convey polish. Even though the app runs primarily in the menu bar, any popover or window it shows should look cohesive and well-crafted (no debug or default look). Icons used (for the app's menubar icon, any within the UI) should be custom or SF Symbols that match the style – giving an impression of a professional tool.

- **Menu Bar Application (Primary Interface):** The app will run as a **menu bar extra**, meaning it presents an icon in the macOS menu bar at the top of the screen rather than a persistent Dock icon (the app can have `LSBackgroundOnly` or use the new SwiftUI `MenuBarExtra` scene for implementation). The menu bar icon should be visible at all times when the app is running, allowing quick access. Clicking the icon will open a **popover** or drop-down panel that contains the main interface (timeline view and controls). The menu bar icon should be a simple, monochrome icon (to match macOS menu bar style) that possibly hints at a timeline or clock. The menu extra should also have a standard menu structure accessible (for secondary click or via a small arrow in the popover) with items like **"Preferences…", "Quit"**, etc., to manage the app. The menubar approach ensures the app is unobtrusive – it doesn't demand screen space until needed, but is always a click away for quick review or inserting a marker.

- When the user clicks the icon, the pop-up should appear below the icon. The popover should be sized reasonably (not fullscreen, ideally something like 400-500px width, and height enough for content but not overly large on small screens).
- The menu bar app should **auto-start on login** if the user allows. Provide a setting (in Preferences) like "Launch at login" which, when enabled, registers the app to start up automatically. This ensures continuous tracking without the user having to manually launch it every reboot.
- The app should run continuously in the background, consuming minimal resources (see Performance below). Even if the UI popover is closed, the background logging should continue.

- No Dock icon should appear (unless launched explicitly for preferences). Implement as a proper UIElement or use `MenuBarExtra` (Ventura and later) which automatically has no Dock item. If older OS support is needed, use the NSApplication API to opt out of dock and have an NSStatusItem.

- **All-Local Storage & Configuration:** The application must store **all data locally**. This includes the activity log database, user preferences, and rule definitions:

- **Database:** Use a local SQLite file as discussed (no remote DB or cloud sync). Ensure file writes are atomic to avoid corruption (SQLite by default will handle this with journaling).
- **Configuration Files:** If any configuration is needed (like the rules in YAML, or an ignore list, etc.), these should reside in the user's Library under the app's sandbox/container or Application Support directory. For example, `~/Library/Application Support/TimelineTracker/rules.yaml` (assuming "TimelineTracker" as the app's name placeholder). These files should be plain text (for rules) or plist (for user settings, though preferences can also be stored with UserDefaults).
- The user's data should be easy to locate and inspect if they desire. In the UI's privacy section, we will display the path to the data file and offer a button like "Reveal Data File in Finder" for transparency.
- **Local Rule Files:** Storing rules in a flat file (YAML/JSON) as specified is important for transparency and easy editing. The app should monitor this file for changes (if user is editing externally) or at least reload rules on app launch and when the user explicitly saves changes via the UI. The rules file is part of the local config – no part of it goes to a server.

- The app should **not create or use any temporary internet cache** or send any info to system logging beyond what's necessary. In other words, all significant data remains in the confines of user-controlled storage.

- **No Networking or Telemetry:** The application will perform **zero network requests** and collect **no telemetry** beyond local logging:

- No usage analytics, crash reports, or update pings should be sent automatically. (If crash reporting is desired for development, it must be strictly opt-in and local, but by default, nothing leaves the machine).
- The app should not even check for new versions by itself. At most, it can have a manual “Check for updates” that opens a webpage or instructs the user, but automatically phoning home is not allowed by this requirement.
- This also means the app should not include any 3rd-party SDKs that might silently send data (e.g. avoid marketing analytics frameworks). The product’s value proposition includes privacy, so we must ensure compliance by design.

- If the app needs to display any online content (unlikely in this context), it must be explicitly approved in design. But as of this spec, there is no feature that requires internet. Even for help documentation, consider bundling it or linking to a user’s web browser rather than fetching in-app.

- **Performance and Efficiency (Background Operation):** The app should be highly efficient in terms of CPU, memory, and power usage, as it will run continuously in the background.

- **CPU Usage:** The logging process must be lightweight. Responding to system events (app switches, etc.) is cheap, but if we use polling (e.g. to detect window title changes or idle time), it should be done at a reasonable interval and not consume noticeable CPU. Aim for near-0% CPU when the user is not actively switching apps. Even during active use, the operations (writing a DB entry, updating UI if open) should be very fast. Apple Silicon is very efficient, but we must still avoid unnecessary loops or busy-waiting. Use timers, system callbacks, or run-loop sources properly.
- **Memory Usage:** Keep the resident memory footprint low. Storing the entire log of a year in memory is not necessary – use database queries to fetch only what's needed for display. If using SwiftUI, be mindful of retaining large data structures. Aim to keep the app well under 100MB of RAM while idle (likely much less, as it’s a simple utility). Use lazy loading for UI components (don’t load stats data until needed, etc.).
- **Long Uptime Stability:** The app might run for days/weeks without restart. It must remain stable (no memory leaks or accumulative CPU usage over time). Test with long-running scenarios. If using timers for polling, ensure they don’t accumulate if not handled. Use appropriate background task APIs if needed.
- **Background Mode:** As a menu bar app, it will always run in the background. Ensure that even if no UI is open, logging still happens. If using SwiftUI `MenuBarExtra`, the app logic might live in an `App` struct – ensure the logging starts on launch (perhaps using an `@main` App that sets up a background Task or an ObservableObject that begins logging).
- **Power Efficiency:** Especially for MacBooks on battery, the app should not significantly impact battery life. Utilize Apple’s power APIs if needed (e.g. `ProcessInfo.performActivity` with `.background` if doing background work, to let system optimize, or simply ensure low usage). Idle-detection polling can perhaps be done using a system event instead of frequent checks (e.g. subscribe to screensaver start as an indicator of idle, etc.). If polling every second is needed for timely logging, consider whether that could be increased to every few seconds when on battery. However, user idle detection likely can be done via event (the system’s idle timer or IOKit) which is efficient.
- **Responsiveness:** The UI should load quickly when invoked. Clicking the menubar icon should show the popover immediately (under ~0.5s). Scrolling the timeline or switching to the stats view should be smooth. Heavy computations (like summarizing a year of data) should be done off the main thread to avoid UI freezes – possibly using background threads or pre-computation. If a computation is expensive, show a quick loading indicator rather than hanging.

- In summary, the app’s presence should be practically unnoticeable in terms of resource usage until the user actively opens it to interact.

- **Optional Data Encryption:** Provide an **optional encryption toggle** for users who want an extra layer of security for the stored activity data. If feasible, implement encryption for the SQLite database:

- When encryption is **enabled** (through a setting in the Privacy preferences), the app should encrypt the database file on disk. This likely means using SQLCipher or another encryption mechanism, since default SQLite doesn’t encrypt by itself. Alternatively, the app could store the DB on an encrypted disk image or use macOS Keychain-secured file encryption.
- The user should be prompted to set a **passphrase** (or use their macOS user password or iCloud key, etc.) when enabling encryption. That passphrase will be required to open the database. The app can store the key securely in the user’s keychain (so they aren’t prompted each time, unless we want maximum security where they enter it each launch).
- If encryption is on, all sensitive data (app names, titles, etc.) in the DB should be encrypted at rest. The app will decrypt on the fly when reading for display.
- This feature is optional because it adds complexity – some users might skip it. It should default to off (unencrypted DB, relying on OS file system security). If off, the DB is a plain SQLite accessible by the user or anyone with disk access. If on, it’s encrypted and unreadable without the key.
- **Feasibility:** Using SQLCipher (an open-source SQLite extension for encryption) is one approach. We must bundle it or otherwise include encryption. If time/complexity doesn’t permit full encryption, this feature could be deferred or clearly documented as experimental. But in the PRD we assume it’s feasible and include it.
- The UI for this will be a simple toggle or button in Privacy settings: e.g. “🔒 Encrypt Activity Data”. Toggling it ON will prompt for a password (with confirmation). Toggling OFF (decrypting) might not be offered; we could say once encrypted, it stays that way unless the user exports and re-imports data. However, ideally allow turning off by asking for password to decrypt and save plain DB – but that’s additional complexity, so can be optional. For now, assume one-way enabling is fine (or treat off similarly by re-writing the DB unencrypted).
- Ensure the encryption/decryption process is done safely (e.g. copy data to a new encrypted DB file, then replace, rather than encrypting in-place which is risky).

- If not feasible to implement encryption, at least the design allows it. (But for no ambiguity, we plan as if implementing it.)

- **Security and Privacy Compliance:** (Some of this overlaps with privacy, but as non-functional aspect) The app must comply with macOS security requirements:

- It should have the correct usage descriptions in its Info.plist for any privacy-sensitive APIs (like Accessibility or Screen Recording). e.g. NSAccessibilityUsageDescription, NSScreenCaptureUsageDescription, etc., explaining why we need those (e.g. “Needed to log active application and window title for timeline tracking.”).
- The app should guide the user to enable the required permissions (via System Settings) on first launch or whenever needed. If permissions aren’t granted, the app should detect this and show an informative message – e.g. “Please enable Accessibility access for Timeline Tracker to record application usage” with a button to open System Preferences to the right pane.
- All data is stored locally as described, and no unauthorized access should occur. If the user has encryption on, ensure the key is not stored plainly. If off, at least rely on OS-level protections (file permissions – data should be stored in the user’s directory, not world-readable).

- The app will ignore any data it's not explicitly supposed to record. For example, it will not capture keystrokes or screenshot contents – only app names and window titles. We ensure not to log anything sensitive beyond window titles, and if window titles themselves could be sensitive (they sometimes contain document names), we provide the user tools like the **ignore list** (see Privacy section) to exclude specific apps from logging. This is a privacy requirement.

Data Structures (Database Schema & Configuration)

All data is stored locally in a SQLite database (with possible configuration in external files). The following schema defines the tables and data structures used by the application:

- **Activity Log Table (`Activities`):** This table stores each continuous activity session (the basic unit of the timeline). Each record represents a time interval during which the user was doing a particular thing (using a specific app/window, or idle).
- **Columns:**
    - `id` – *INTEGER PRIMARY KEY* (auto-increment unique identifier for each activity record).
    - `start_time` – *DATETIME* (timestamp for when this session began. Stored as Unix epoch in seconds or ISO8601 text, UTC or local time consistently).
    - `end_time` – *DATETIME* (timestamp for when the session ended. For an ongoing session that hasn't ended yet, this could be NULL or equal to start_time until updated).
    - `app_name` – *TEXT* (the name of the application in use during this session, e.g. "Google Chrome", "Slack". Ideally the human-readable app name; could also store bundle ID separately if needed for rules matching).
    - `window_title` – *TEXT* (title of the frontmost window/document when the session started. Example: a browser tab title, document name, or window caption. If the session spans multiple title changes but app stays same, we may either split sessions at title change or update this title – in this schema, we assume sessions are split when title changes, so one session has one title).
    - `is_idle` – *BOOLEAN* (0/1 flag indicating if this is an idle period. For idle sessions, `app_name` might be recorded as "Idle" or could be empty, and window_title might be empty or a constant like "Idle". This flag makes it easy to filter out idle time in queries).
    - `tag_id` – *INTEGER* (foreign key referencing the `Tags` table's `id`, if a rule assigned a tag to this session. Null if no tag or not classified. If multiple tags are allowed, a separate mapping table would be needed, but for simplicity we assign at most one primary tag per session).
    - (Optional) `duration` – *INTEGER* (store the duration in seconds for convenience, though it can be computed from end-start. Could be a generated/virtual column in SQLite or updated on session end).
    - (Optional) `app_bundle_id` – *TEXT* (to store the bundle identifier, e.g. "com.apple.Safari", in case rules want to match on bundle id or for more precise app identification. Not strictly required since name is usually enough).
- **Description:** This table is the core log of what happened when. Each row tells us "User spent [end_time - start_time] duration from start_time to end_time on app X (window Y) and it was tagged as Z (if any)." Idle sessions will appear here as well (with is_idle = 1). Manual markers are *not* in this table (they are separate, see below). This table will be frequently appended to (when new sessions start or end) and read for generating the timeline and stats.
- **Indices:** We should index `start_time` (and possibly `end_time`) for time-range queries (to fetch a day's or week's data quickly). Also index `tag_id` if we often query by tag (e.g. total time per tag). Index `app_name` if queries per app are needed (or better, if we had an App table with IDs, but not necessary). A composite index on (start_time, end_time) can help range selects.

Ensure foreign key constraint on tag_id (with cascading updates if tag name changes, etc., or just static if tag changes won't retroactively apply).

- **Relationships:** Many-to-one with `Tags` (each activity can have at most one tag). If multiple tags per activity were desired, a join table (ActivityTagMap) would be needed, but that's outside current scope.

- **Segmented Sessions:** *Note:* In our design, the `Activities` table above already represents segmented sessions (merged contiguous periods). We may not need a separate "segments" table if we treat each row as an already-merged segment. However, the spec listed "raw activity logs" vs "segmented sessions". Interpreting that:

- We might choose to maintain a **Raw Events Table** (`ActivityEvents`) separate from the `Activities` sessions. Raw events could log every little event: app gained focus, title changed, etc., with just a timestamp and description. Then a processing step merges them into `Activities`. This could be useful for debugging or if we want to regenerate sessions with different rules.
- **Raw Events Table (optional):** If implemented, columns might be: `id`, `timestamp`, `event_type` (e.g. "app_focus", "title_change", "idle_start", "idle_end", "marker"), `app_name`, `window_title`, `note` (for marker text if event_type is marker). This table would append an entry whenever something happens. Then logic would construct sessions for the Activities table.
- For simplicity, we can also consider the `Activities` table to be built directly without storing raw events separately. Many time-tracking apps do not store the raw switch events once they've computed the session.
- **Decision:** We will treat `Activities` as the primary storage of timeline data (already merged), and not maintain a separate raw log table to avoid duplication. If needed, we log some low-level events in-memory just to handle merging logic.

- Therefore, **Segmented Sessions = Activities table rows**. There is a 1:1 mapping conceptually: each row is a segment in the timeline.

- **Manual Markers Table (`Markers`):** This table stores user-inserted markers/notes.

- **Columns:**
  - `id` – *INTEGER PRIMARY KEY*.
  - `timestamp` – *DATETIME* (when the marker was added; this is a point in time, not an interval).
  - `text` – *TEXT* (the note content the user entered, e.g. "Lunch break").
  - (Optional) `tag_id` – *INTEGER* (if we want to optionally associate a tag with the marker or treat it as a category of activity – usually not needed, the text suffices).
  - (Optional) `linked_activity_id` – *INTEGER* (if we want to link a marker to a specific activity session. Not required, but for example if a marker is meant to label the idle time that follows it, we could store that relation. However, the UI can infer context by timestamp, so we likely won't use this.)
- **Description:** Each entry is an independent note the user added at a particular time. Markers do not have duration. In the UI timeline, they will appear at their timestamp (for instance as a line or icon with the note text). Markers may coincide with a period of idle or active use, but they live in their own table for simplicity.
- **Indices:** Index on `timestamp` (for quickly finding markers in a time range to display on timeline). Possibly full-text index on text if we add search, but not necessary now.

- **Relationships:** No mandatory foreign keys. If `linked_activity_id` were used, that would reference `Activities.id`.

- **Tags Table (`Tags`):** This table defines the set of tags (categories) that can be assigned to activities via rules.

- **Columns:**
  - `id` – *INTEGER PRIMARY KEY*.
  - `name` – *TEXT* (tag name, e.g. "Communication", "Development", "Entertainment". Should be unique).
  - `color` – *TEXT* (store a color code for the tag, e.g. a HEX color string or a predefined color name. This is used to color-code the UI for this tag. If not needed, can be NULL or a default color assigned. Including it enhances UI clarity).
  - (Optional) `icon` – *TEXT* or *BLOB* (if we allow an icon for a tag, e.g. a SF Symbol name or image blob. Probably overkill; not required).
- **Description:** This table contains the master list of tags available. It's basically the unique labels that can be applied. The user can manage tags through editing rules (tags might be implicitly created when a new tag name is used in a rule, or we might have a tag management UI).
- Initially, we might not have a separate UI to manage tags aside from rules – i.e. tags get created as needed by rules. But storing them in a table allows consistency (e.g. if multiple rules output the same tag "Project X", there's just one tag entry with that name).
- **Indices:** Unique index on `name` (to prevent duplicates). The `id` is used in the Activities table for quick joins.
- **Relationships:** One-to-many with `Activities` (a tag can label many activity sessions). Also, one-to-many with `Rules` (as each rule assigns one tag – although we might not explicitly link those in DB, since rules might be file-based).

- **Note on Tag usage:** A session can have at most one primary tag in this design. If future needs require multi-tagging, we'd need a join table, but for now we treat tags as categories (one category per session). The UI will use tag names and colors extensively (timeline segment coloring, stats grouping, etc.).

- **Rules Storage:** The rules themselves will be stored in a user-editable text file (YAML or JSON). We will parse this file and use it to drive the tagging logic. We do not necessarily need a dedicated database table for rules, but for completeness we describe the structure:

- **File:** `rules.yaml` (or `rules.json`) located in the app's Application Support directory (or a subdirectory `Rules/` or similar). YAML is preferred for readability.
- **Format:** The file will contain an array/list of rule definitions. Each rule has properties such as:
  - `name` (optional human-friendly name/description of the rule, e.g. "Tag Slack as Communication").
  - `app` or `app_name` condition – could be a string or list of strings. This matches against the application's name (or bundle ID). If multiple apps are listed, the rule matches if any of them match (logical OR within this field).
  - `title_contains` condition – a string or list of strings to search for in the window title (case-insensitive containment). If a list, any match suffices (OR within the list). We may also support regex via a separate field like `title_matches_regex` if needed.
  - (Optionally, other conditions in future like `title_not_contains` or time-of-day constraints, but not in initial scope).

    - `tag` – the tag name (string) to assign if the conditions match. This should correspond to a tag in Tags table. If the tag does not exist yet, the app will create a new tag entry for it (with a default color).
    - `priority` – number indicating priority (higher = higher priority). Could be 0-100 or any range; we'll just compare numerically.
    - `enabled` – boolean (if the user wants to temporarily disable a rule without deleting it).
    - Example (YAML):

    ```yaml
    - name: "Slack = Communication"
      app_name: "Slack"
      tag: "Communication"
      priority: 5

    - name: "IDE work = Development"
      app_name: ["Xcode", "Android Studio", "Visual Studio Code"]
      tag: "Development"
      priority: 3

    - name: "Project X in Title"
      app_name: "Google Chrome"
      title_contains: "Project X"
      tag: "Project X Research"
      priority: 4
    ```

    In the above example: any Slack usage gets "Communication". Using Xcode/VSCode etc gets "Development". Chrome with "Project X" in the title gets "Project X Research" tag. If Slack also had "Project X" in title for some reason, Slack rule (priority 5) would win over Project X rule (priority 4).
- **In-Memory Representation:** On launch, the app will read this file and parse into a rules list (objects). The rules engine then iterates through or uses a matching algorithm to apply tags to each new Activity session (likely check each rule in order of priority).
- **Editing:** When user updates rules via the app's editor UI, the file should be overwritten or updated, and the new rules should be reloaded (and if chosen, reapply to recent un-tagged sessions).
- We will **not** store rules in the SQLite DB by default (to keep them easily editable and because rules are more configuration than data). This also allows the user to back up or share the rules file separately from the activity data.
- If needed, we could provide a command to export the rules to JSON or import from JSON (trivial since it's text anyway).

- **Validation:** The app should handle parse errors gracefully (e.g. if the YAML has a syntax error, show an error message in the UI and do not wipe the existing rules in memory until fixed).

- **Ignore List:** A small but important data set is the list of apps to ignore (not track). We mention it here for completeness:

- This can be stored either in a simple text config (e.g. an array in the YAML, like `ignore_apps: ["App1", "App2"]`), or in a separate preferences store.

- Alternatively, we can repurpose the rules system: e.g. a rule could have a special tag "_ignore" meaning skip logging. But simpler is to implement ignoring at capture time (we won't log events for those apps at all).
- Implementation: We'll maintain an **Ignore List** (maybe in UserDefaults or a config file). This list contains application identifiers (preferably bundle IDs to be precise, or app names if easier).
- The logger will check the frontmost app against this list – if it's on the list, the logger will **not record that app's usage**. Effectively, time spent in an ignored app will either appear as a gap or could be lumped into idle. We choose to **leave a gap** (i.e., do nothing; do not log idle either because the user was active but in an ignored app).
- The effect is that the timeline will not show anything for those intervals (meaning the timeline might jump over that period or show it as unlabeled time). We can clarify in UI (privacy section) that ignored apps' time will simply not be recorded or displayed.
- Data structure wise, the ignore list could be just a list of strings in a config. We don't need a DB table unless we want to track them like tags, but it's simpler not to.
- Possibly include an `IgnoredApps` table for completeness? That might be overkill. We can just store in preferences.

In summary, the data storage design emphasizes a single source of truth for time logs (Activities table), a separate table for user markers, a mapping of tags, and an external definition of rules. All these are local. Below is a simplified **ER diagram** in words: - Activities (many) —(tag_id)-> Tags (one) [optional relationship] - Activities (many) — contains -> [app_name, window_title, times] (not a separate table for apps, using text to simplify). - Markers (many) standalone (each marker has timestamp, text). - Rules (many) in file — assign -> Tags (one per rule). - IgnoreList (conceptual, not a DB entity, just used in logic).

This schema should be implemented in the SQLite DB on first run (the app should create tables if not existing). Migrations should be handled if schema changes in future (e.g. adding a column for something). All data is local and relational, enabling complex queries if needed (like "How much time on Tag X between date1 and date2"). The developer must ensure to use parameterized queries to prevent any SQL injection issues (though there's no external input except maybe rule text, but we handle that carefully).

Rules Engine

The **Rules Engine** is responsible for automatically categorizing activity sessions based on the user-defined rules. This section describes how rules are applied and managed:

- **Rule Matching Process:** Every time a new activity session is recorded (or when an existing session's details are finalized, like end time), the rules engine should attempt to match it against the set of rules:
- The engine will evaluate each rule on the session's attributes. A rule is considered a **match** if **all specified conditions** in that rule are satisfied by the session. Conditions include:
  - **Application Name condition:** If the rule specifies an `app_name` (or multiple), then the session's app_name must exactly match one of them. This comparison should be case-insensitive and ideally match the full app name or bundle ID. We might allow wildcard matching (e.g. "Visual Studio*" to catch "Visual Studio Code"), but exact or list of exact names is simplest.
  - **Window Title condition:** If the rule specifies `title_contains` (one or more substrings), check if the session's window_title contains any of those substrings (case-insensitive). If the rule specifies a regex pattern (in a `title_matches` field, for example), test that regex on the window_title. (Regex support is advanced; for initial version, substring match is sufficient).

  - A rule can have both app and title conditions; in that case **both must be true** (logical AND). If a rule has only an app condition or only a title condition, that alone can trigger a match.
  - If a rule has multiple values for a field (like multiple app names), that field condition is true if any value matches (OR within that field).
  - We will not implement complex nested logic in the first version (no explicit NOT or XOR etc.). The user can achieve an exclude by making a higher priority rule to tag something differently or ignore, etc.
- If a session matches a rule, the rule's `tag` is considered for assignment to that session. However, we must consider priority:
  - The engine should evaluate all rules that match, then choose the one with the **highest priority** number as the winner. Only that rule's tag will be applied.
  - If exactly one rule matches, use that rule's tag.
  - If none match, the session remains untagged (or tagged as a default like "Uncategorized" if we want to have a tag for it).
  - If multiple match with equal priority (and that priority is the highest), use the one that appears last in the rules file (as a deterministic tie-breaker). This means the rules file order can implicitly resolve ties if needed (document to user if necessary).
  - Once a tag is assigned, it should be stored with the activity record (as `tag_id`). If the session is still ongoing (not ended yet), we can assign a tag immediately based on the starting window title. If the window title changes mid-session, actually that would end the session and start a new one under our logging logic, so we're fine. So tag likely remains consistent through the session unless the user changes rules later.

- The tagging should happen quickly and ideally in the background thread so as not to delay logging. However, since the number of rules is likely small (tens or a hundred at most) and each check is a few string comparisons, performance is not an issue doing it on the fly.

- **User-Editable Rules Format:** As described in Data Structures, rules live in a YAML/JSON file. The user interface will allow modifying them, but here we clarify the expected format to ensure unambiguous implementation:

- We will use **YAML** for the rules file (for readability, and it can easily represent lists of conditions). JSON could be used, but YAML allows comments and is more user-friendly for config.
- The rules file is essentially an array of rule objects. For each rule, the possible keys are:
  - `name` (string, optional) – just a label for the user's reference.
  - `app_name` (string or list of strings, optional) – application match condition. If absent, it means any app is fine (so rule might rely on title only).
  - `title_contains` (string or list of strings, optional) – window title substring condition. If absent, means any title (so could match just on app).
  - `tag` (string, required) – the name of the tag to assign if rule matches.
  - `priority` (int, optional but recommended) – priority value. If not provided, default to 0 (lowest). To avoid ambiguity, the app might enforce that each rule has a priority explicitly set when editing.
  - `enabled` (bool, optional) – default true; if false, rule is skipped (this allows user to toggle rules without deletion).
  - (We will ignore other potential fields in initial version.)
- **Parsing:** The app must parse this file at startup or when updated. Use a YAML library or manual parsing. Validate each rule entry for correctness (e.g. tag present, priority is number, etc.). If any rule is malformed, skip it and maybe log a warning (in a log file or console).

- **Storing Tags:** When reading rules, for each unique `tag` string, ensure it exists in the Tags table. If not, create a new tag entry with a default color (perhaps generate one deterministically or random pastel color). If it exists, reuse it. This ensures consistency between rules and the tags repository. The user can later change tag names via rules – if a tag name is changed in the rules file, we could either update the corresponding tag entry's name (and thus all past sessions labeled with it), or treat it as a new tag separate from old (which might orphan old entries under an old tag name).
    - To keep it simple: if user changes a tag name in rules, we will create a new tag entry for it, and past sessions with the old tag name remain as they were. The user can merge them manually if desired by reassigning or editing data. We won't auto-retroactively change historical data's tag names, since that could be confusing. Alternatively, we could identify that the old tag has no rule now and leave it. This is a design decision; we choose to keep historical data unchanged by default.
- **Example YAML Rule:** Here's a concrete example to illustrate format and what it means:

```yaml
- name: "Work – Coding"
  app_name: ["Xcode", "IntelliJ IDEA", "Visual Studio Code"]
  title_contains: ["ProjectAlpha", "ProjectBeta"]
  tag: "Coding"
  priority: 8

- name: "Emails"
  app_name: "Mail"
  tag: "Email"
  priority: 5

- name: "Web Meeting"
  app_name: "Google Chrome"
  title_contains: "Zoom Meeting"
  tag: "Meetings"
  priority: 7
```

  In this example, the first rule tags sessions as "Coding" if the app is one of several IDEs **and** the window title contains either "ProjectAlpha" or "ProjectBeta" (perhaps filenames or window titles containing those project names). The second rule tags any usage of the Mail app as "Email". The third tags any Chrome usage where the title has "Zoom Meeting" (maybe the user's using Zoom in a browser tab) as "Meetings". The priorities indicate "Work – Coding" (8) will trump "Web Meeting" (7) if something somehow matched both, and both trump "Mail" (5) but that's separate domain.
- **Applying Rules to Idle:** By default, idle sessions likely remain untagged (or get a fixed tag like "Idle" if we add that as a tag entry). We might not allow rules to target idle (except maybe by app_name "Idle", but that's artificial). It's safe to treat idle as its own category outside user tagging (or the app can auto-tag idle with "Idle" category internally).
- **Updating Rules at Runtime:** When the user changes rules (via the UI editor), the app should:
    1. Save the new rules to the YAML file.
    2. Reload/parse the rules into memory.
    3. Rebuild the Tags table if needed (add any new tags).
    4. **Optionally** ask the user if they want to re-classify past data with the new rules. If yes, iterate through Activities and update tag assignments. This could be time-consuming for large histories, so it might be better to only apply going forward by default, to avoid

freezing the app. Perhaps provide a “Reapply to existing data” button separate from just saving rules, so the user can choose when to do it. This operation would involve scanning each activity and applying the new rule logic – since data is local, it’s feasible.
5. Ensure new rules take effect for any new sessions immediately.

- **Performance:** The number of rules is expected to be moderately small (maybe 10-50 on average, could be 100+ for power users). The checking overhead is trivial. We should ensure string matching is done in an efficient manner (e.g. lowercase the strings once, or even precompile regex if used). But given the scale, even brute force checking is fine.
- **Edge Cases:** If a rule’s tag points to something that user later deletes from Tags (if we had tag management separate) – not likely since we derive tags from rules. If an activity already tagged gets retroactively no rule (because user removed that rule), that activity retains its tag in the database. We might treat it as now “manual” classification or just leave it as historical with a tag that’s no longer produced by any rule. This is acceptable. We won’t delete tags just because rules removed them, unless user explicitly purges unused tags.

- The rules engine should **ignore disabled rules** or any that are clearly not applicable. And importantly, it should not assign tags for ignored apps (the ignore list likely prevents logging those sessions entirely, but if an ignored app’s session somehow exists, perhaps do not tag it).

- **Manual Overrides:** The scope of this PRD doesn’t explicitly mention manual categorization by user (except via markers). Some apps let you manually reassign a category for a specific session. We have not included a UI for manual tag override, but it’s worth noting. If in future needed, one could double-click a session and change its tag. That would effectively override the rule engine for that entry. Since we aim for minimal ambiguity: currently, we won’t implement manual tag editing; categorization is solely via rules. The user can achieve a manual effect by creating a specific rule for that single case or by splitting tasks using markers. We mention this just to clarify expected behavior: no manual post-hoc tagging in v1.

In summary, the rules engine uses the user-defined rules to map low-level usage data to meaningful categories. It relies on the rules file, which is maintained by the user (with app assistance). It must be reliable and predictable (no randomness). The developer should ensure the engine is thoroughly tested with various scenarios to verify that the highest priority rule indeed wins and that rules combination logic (AND/OR) works as intended. Also test that updating the rules file live has the intended effect. The rules system is a centerpiece for the “smart” aspect of the app, turning raw logs into useful insights, so it must be robust.

User Interface Specification

The application’s UI will be primarily exposed through a menu bar popover and a few dedicated windows/panels for specific tasks (like Preferences, which include Rules Editor and Privacy info). The design must be intuitive and require minimal clicks for the main use-cases (view timeline, add marker, check stats). Below, we describe each major component of the UI and its requirements:

Menubar Icon and Popover (Main Interface Container)

- The **menubar icon** (in the system status bar at top of screen) is the entry point. It should be a simple icon (e.g., a small clock or timeline graphic) that appears alongside other status icons. Clicking this icon toggles the **popover** which contains the main UI.
- **Popover Behavior:** The popover will show when the icon is clicked, and hide when clicking outside or when the icon is clicked again (typical menubar behavior). The popover should also be able to be pinned if Apple’s API allows (not critical).

- The popover acts as the container for either the Timeline view or the Stats view. We will provide a way to switch between "Today's Timeline" and "Statistics" from within the popover (for example, a segmented control at the top, or a toolbar with two toggle buttons "Timeline" and "Stats").
- At the top of the popover, we can display a **header** with the app name or icon and the current date, or simply a toggle for views. For example:
- A segmented control: `[ Timeline | Stats ]` that switches the content below.
- The current date could be displayed when in timeline view as context (e.g. "Today – Mon, Jan 12").
- The popover size: width ~400px (depending on content) and height ~500px (scrollable if content exceeds). It should be resizable if necessary, or at least adjust height to content up to a max.
- **Themeing:** The popover should support dark mode (likely by default SwiftUI components do). Also consider using vibrancy/material effect for background to match macOS style for menus (perhaps `NSVisualEffectView` behind SwiftUI or SwiftUI's `.background(Material.thin)` etc., but ensure text remains legible).
- The menubar menu (if user right-clicks or if we provide a dropdown arrow) should include at least:
- "Preferences…" (opens the Preferences window containing Rules & Privacy).
- "Quit [AppName]".
- Possibly "About [AppName]" if not included elsewhere.
- If auto-launch is not set through system, maybe a "Launch at Login" toggle here or in Preferences.
- These can also be provided as buttons within the UI, but a standard menu is good for quick access.

Timeline View (Daily Activity Timeline)

This view shows a detailed timeline of the user's activities for a selected day (defaulting to today). It is the core view for daily review.

Layout and Content: - The timeline should be displayed as a **vertical scrolling list** of time-stamped entries, grouped by hour or sequentially sorted by time. Each entry corresponds to an activity session (from the `Activities` data). - Each activity entry in the list should display: - The **time interval** (start–end) of the session (e.g. "09:00 – 09:50" or "09:00 – 9:50 AM" in a user-friendly format). - The **application name** and possibly an icon for the app (we can get the app's icon via NSWorkspace if allowed, but to keep UI clean, text might suffice. A small 16x16 icon next to name could be a nice touch). - The **window/ document title** (possibly truncated if very long, with tooltip for full title). For example: "Google Chrome – *Stack Overflow - Question*". - The **tag/category** if one is assigned: This can be shown as a colored label or badge next to the entry. For instance, if tagged "Development", either display a colored dot or a pill with the tag name. The color comes from the tag's defined color. If no tag, it could show nothing or a default like "Uncategorized" (in a neutral color). - Possibly an icon or marker if this entry has a manual note overlapping or if the user manually annotated it (markers are separate but see below). - **Idle segments** should be clearly indicated, so the user knows when they were inactive. For example, an idle entry might say "09:50 – 10:00 – Idle" in a lighter or italic font, with maybe a pause icon. The user might later add a marker "Meeting" at 09:50 to label that idle time; in that case we show the marker (but still the entry is Idle). - **Manual Markers display:** Markers that fall within the day should appear in the timeline at the appropriate point. They are instantaneous points in time, not intervals, so how to display: - If a marker falls in the middle of an activity entry, we can show it as a sub-item or overlay. For example, at 11:00 user added "Call with Bob". The timeline at 11:00 might show something like a small timestamped line: "→ 11:00 Marker: Call with Bob" (perhaps indented or with a pin icon). This could be listed between entries or on top of the corresponding entry. Alternatively, show a small pushpin icon on the activity entry's line with that time – but likely listing it as its own row with the time. - If a marker coincides exactly with the start or end of an activity, it can be merged in display (e.g. "09:50 Idle (Marker:

Lunch break)”). - Keep it simple: every marker is an entry in the timeline as well, perhaps styled differently (e.g. no duration, just a single time). Possibly a different color or icon (like a star or note icon). - **Grouping by Day:** Since this is the daily view, by default it shows the current day (from 00:00 to 23:59). We should allow the user to change the day: - Perhaps a date picker or left-right arrows to go to previous/next day. For example, at the top of the timeline view: “< Jan 11 | Today (Jan 12) | Jan 13 >” to navigate. Or a calendar icon that opens a date chooser. - The UI should update to show the selected day’s data. If a day has no data (in the past before app installation or future), show no entries or a message “No data”. - The timeline should also indicate the total active time vs idle for that day somewhere (e.g. a footer: “Active: 7h 30m, Idle: 16h 30m”). Or this can be in Stats view; but it’s useful at daily level too. - **Interactions:** - Scrolling: The user can scroll through the day’s timeline. If the day is long with many entries, the list might be tall, but that’s fine. - Hovering an entry could show more detail (full window title, etc.). - Perhaps clicking an entry could expand it (if we want to show sub-info like exact tag, or allow editing). We might allow right-click on an entry to do things like “Edit Tag” or “Add Note” in future. In v1, we might not implement editing, so clicking can just do nothing or highlight. - Deleting entries: Not a primary feature, but if needed, a right-click “Delete entry” could remove that log (and from DB). We don’t necessarily encourage deletion since it affects data integrity, but a user might want to delete something sensitive (alternatively they should have added that app to ignore list to not log it). We can skip deletion for now. - **Add Marker:** The timeline view could incorporate the ability to add a marker at the current time easily (maybe a button at top “Add Marker”). However, since we already have a global shortcut and menu option, the timeline itself might not need an extra control for that. - If the user does press the global shortcut while the popover is open, perhaps we show the marker input UI (detailed next) overlaying the timeline.

- **Design considerations:** Use a **List view** in SwiftUI or equivalent. Each row can be a HStack with time and details. Use clear typography: e.g. time in bold or a monospace for alignment, app name normal, title in quotes or smaller font. Tag can be a colored capsule on the trailing end. Idle entries could be gray/italic. Markers could be centered or a different symbol.
- Possibly visually differentiate consecutive entries by background shading (alternate row colors or a subtle separator with time labels).

- Show hour separators maybe (like grouping by hour blocks), but that might clutter; possibly skip, since each entry already shows time.

- **Example of Timeline UI (textual):**

```
Today (Mon Jan 12, 2026)
08:45 – 09:00  Slack — #general (Tag: Communication)
09:00 – 09:50  Xcode — MyAppProject (Tag: Development)
09:50 – 10:00  **Idle** (no activity)   [Marker: Standup Meeting]
10:00 – 10:30  Google Chrome — Docs: PRD Specification (Tag: Writing)
10:30 – 11:00  Mail — Inbox (Tag: Email)
11:00 – 12:00  **Idle** (no activity)   [Marker: Lunch break]
...
Active time: X hours, Idle time: Y hours.
```

The above shows how a marker (Standup Meeting) at 09:50 is during an idle period, and appears annotated next to it. Lunch break marker at 11:00 for an idle session from 11:00-12:00, etc. This is just to illustrate.

- The timeline view is the default shown when the user opens the app from the menubar. It allows them to quickly glance at what they've done so far today and add notes if needed. It should update in real-time: if the user stays the panel open, and they switch to another app or a new session starts, the timeline should live-update (append the new entry or update the current entry's end time). SwiftUI can handle dynamic data if bound properly. Ensure that heavy updates (like ticking a timer every second) are avoided; we can update the UI on significant changes (new app or idle state) rather than every second of duration.

Manual Marker Input UI

When the user triggers adding a manual marker (via the menu or hotkey), the app should present a **Marker input dialog**.

- **Triggering:**
- If the user selects "Add Marker" from the menu bar menu or presses the global shortcut, open a small prompt window or popover to enter the marker text.
- If the main popover is open, this could be a text field appearing at the top of the timeline view (shifting content down) with instructions like "Add a note for current time: [text field] (Press Enter to save)". This could be convenient if the user is already looking at the timeline.
- If the app is in the background (user presses hotkey while another app is active), we may need to either (a) show the menu popover and focus a text field, or (b) show a small floating text input panel independent of the popover.
    - The simpler approach: on hotkey, open the main popover (if not open) and show the text field at the top of it.
    - Alternatively, a standalone minimal "marker window" could appear (like a HUD window with just a text field and Save/Cancel). This might be more user-friendly if they just want to quickly jot and continue.

- We will implement it as part of the menubar popover for simplicity (if possible to programmatically open the popover and focus a field).

- **Design:** The marker input UI should be minimal:

- A single-line text field (or multiline if note can be long, but likely short notes, so single line is fine) for the note content.
- Placeholder text like "What are you doing? (Add marker)" to hint usage.
- When the field is focused, maybe show two buttons: "Add" (or just enable pressing Enter) and "Cancel" (Esc or a small cancel button).
- If the user presses Enter or clicks Add, the current timestamp is recorded and the text is saved as a new Marker in the database. The UI then clears the field and either hides (if it was a separate window) or collapses it (if integrated).
- If Cancel, simply abort (no marker created).

- The timestamp used should be the moment the user invoked the action (not when they finish typing, ideally, although a few seconds difference is negligible). Implementation: probably take timestamp on opening the dialog or on pressing save. Either is fine; it's usually at that moment anyway.

- **Integration with timeline:** After adding a marker, the timeline view (if showing that day) should immediately show the new marker entry at the appropriate spot. If the timeline is for today, it would appear in order. Possibly scroll to it or highlight it briefly to confirm addition.

- The marker input should not allow excessively long text (maybe limit to ~100 characters) to keep data reasonable.
- If the user wants to mark an earlier time (like forgot to mark something 10 minutes ago), our current design does not allow specifying a custom time – it always uses now. That is acceptable for v1 (the user can at most add a note for now, or they could manually adjust after insertion by editing the DB but that's out of UI scope). So no UI for custom timestamp (that would complicate things).
- **Global hotkey registration:** On app launch, register a global shortcut (e.g. using Carbon or the newer API for hotkeys). If it conflicts or fails, alert the user or allow them to change it in preferences.
- The preferences should allow customizing the hotkey (not mandatory, but good UX). If including, provide a recorder UI in Preferences where user can press a new combination.
- **Edge Case:** If the user triggers a marker while the app doesn't have Accessibility permission, we can still record it since it's initiated by user. That's fine – it's independent of logging context.

Stats Dashboard (Weekly/Monthly/Yearly Summary View)

The Stats Dashboard provides an aggregated view of usage data over longer periods. It should present the information in a visually clear manner, suitable for identifying patterns and totals.

- **View Toggle:** As mentioned, the user can switch to the Stats view via a segmented control or some toggle on the menubar popover. When Stats is selected, the content of the popover changes from timeline list to the stats layout. (If the popover is too small for stats, we might consider opening a separate window for detailed stats. But the requirement suggests including it in the UI, and we can make the popover scrollable or resizable. Alternatively, Stats might be shown in the Preferences window or a separate "Dashboard" window. However, the prompt implies it's a core part of the main UI, so likely accessible from menubar.)
- **Timeframe Selection:** Provide UI elements to select the timeframe:
- Possibly a dropdown or segmented control: "Day / Week / Month / Year / Custom". Given the requirement specifically lists daily, weekly, monthly, yearly, we can have four buttons or a picker.
- When switching timeframe, the content updates to that aggregation.
- For daily, it might show similar info to timeline but in summary form (since detailed daily is timeline, maybe daily here just shows total per tag and maybe a small bar per hour? But we can skip daily summary since timeline covers day details).
- Focus on Week, Month, Year as the main aggregate choices in this Stats view.
- Also possibly allow the user to move between weeks or months (like a < > control or a date picker if custom).
- We could default to "This Week" (Monday to today) or "Last 7 days". Similarly "This Month" or "Last 30 days", etc. Clarify to user via labels.
- For year, likely "Year 2026" (Jan 1 to Dec 31).

- Could allow selecting specific past week/month from a dropdown if needed. Maybe out of scope for initial; showing current and perhaps last might suffice.

- **Content:** The stats view should display:

- **Total time tracked** in that period (excluding idle or including? Could show both).
- **Breakdown by category (tag)** – likely as the primary focus. For example, a list or chart of tags with total hours and percentage. E.g.:
    - Work: 40h (50%)
    - Communication: 5h (6%)

    - Entertainment: 10h (12%)
    - Idle: 25h (32%)
    - (Sum = total hours in period; note that Idle might skew if we always include 24h per day. Perhaps consider active time only by default in these reports.)
- **Breakdown by application** – possibly secondary or on demand. The user might be curious which apps used most. This could be another tab or toggle within Stats ("By Category" vs "By App"). Or a combined view where you might list top 5 apps as well.
    - For v1, perhaps focus on tags since that's what rules allow customizing, which is more meaningful. But showing apps is straightforward too.
- **Visualization:** Use simple charts if possible:
    - A horizontal bar chart for tag totals (bars proportional to time). Could be accompanied by numeric labels.
    - A pie chart for overall category distribution (only if it can be done easily; SwiftUI doesn't have built-in chart prior to iOS 16 unless using Swift Charts framework which is iOS16+/macOS13+. If available, could use it).
    - If not using an actual chart control, we can simulate with colored rectangles in HStack as bars or just text and maybe a progress bar style representation.
    - We should use the tag's color for its bar or pie segment to maintain consistency.
- **Timeline summary:** For weekly or monthly, showing how each day's active hours vary might be useful:
    - For example, a small bar graph where x-axis is days and y-axis is hours active. This gives user an idea of which days they worked longer.
    - Or for a year, show months on x-axis and hours on y.
    - SwiftUI's Chart API (if using macOS 13 or above) can make a quick bar chart for these.
    - If charts are too complex, a simpler approach: list each day with total active hours (like "Mon: 8h, Tue: 7h…").
    - But a visual graph is nicer if possible.

- **Idle handling:** Decide whether to include idle in these stats. Possibly by default, focus on active time, since idle (especially if 24h logging includes overnight as idle) can dwarf meaningful data.

    - We might choose to **exclude idle from percentage calculations** by default. Perhaps show idle separately. For example, "Active time X hours, Idle Y hours (excluded from chart)".
    - Or include idle as just another category (like in example above idle was 32%). But then user's "productive" categories become small portion of 24h.
    - Perhaps better: show both: "Out of 24h: Work 30%, Idle 50%, etc. Active only: Work 60% of active time, etc." This might be too detailed.
    - Simpler: by default consider only active time in the pie/bar charts, but also display total idle time somewhere.
    - We can clarify in UI: e.g. "Active time 50h out of 168h this week (30%). Breakdown of active time: [chart]. Idle time 118h."
    - For now, treat idle separately to give more meaningful breakdown of productive time.

- **Example Stats Output (for a week):**

- Title: "This Week (Nov 1–7, 2026)"
- "Total Active Time: 40h 0m (Idle: 128h)"
- Category breakdown (with colored bars):
    - Development – 20h (50%)
    - Meetings – 5h (12.5%)
    - Communication – 5h (12.5%)

    - Entertainment – 10h (25%)
    - *Other – 0h (if any uncategorized)*.
- [Optionally a bar graph of Mon-Sun active hours].
- Perhaps a toggle or tabs to switch to “By Application”:
    - Then list top apps: e.g. “Xcode: 18h, Chrome: 8h, Slack: 5h, Zoom: 4h, VSCode: 2h, Others: 3h”.
- **Navigation/Interaction:**
- The user might click on a segment in a chart or a tag in the list to filter or see details. For example, clicking "Development – 20h" could highlight those sessions or list which days and apps contributed to that. That’s a possible drill-down feature but not explicitly required. If easy, maybe show a popover with detail or simply not implement to keep scope.
- Ensure the stats update promptly when switching range. Possibly precompute weekly totals to reduce calculation delay, but given local data and SQLite, a query grouping by tag is fast enough for typical data sizes.
- If the user selects a custom date range (if we add that), we should label it clearly and show same info for that range.
- **UI Implementation:** SwiftUI can handle list and basic shapes. We might have to implement a simple Chart manually if not using macOS 13's Charts framework. For example, for each tag we can create a Rectangle with width proportional to time relative to max or total.
- Could also present data just textually if needed, but visuals improve user experience.
- We will aim for at least a bar representation for categories. Possibly using an HStack per tag: colored bar (size = (tag_time / max_tag_time) * some fixed length) plus label and time. Or simpler, a fixed 100% width bar with internal divisions (like a stacked bar per tag) might not be as clear as separate bars per tag.
- Another nice UI element: a donut chart for % breakdown. If using SwiftUI Charts is allowed (macOS 13+), that is trivial. If not, could draw one with paths which is too complex for now. We'll stick to bars.
- The Stats view should give the user a satisfying overview of their habits. It’s effectively a small analytics dashboard for time. The UI should remain uncluttered: likely just one main chart/list and perhaps smaller context info (like daily bars).
- Make sure to label axes or values clearly if charts are used. Use user’s locale for date formatting, etc.

Rules Editor Interface

The Rules Editor allows the user to view and modify the auto-tagging rules in-app, without digging into config files manually. This will likely be in the **Preferences window** as its own tab or section, since it’s more of a settings/configuration aspect than daily use.

- **Access:** The user can open Preferences via the menubar menu (e.g., “Preferences…” menu item). In the Preferences window, there will be a tab or sidebar item for “Rules” (alongside possibly “General” and “Privacy”).

- Alternatively, we could have a direct “Edit Rules” item in the menubar menu that jumps straight to that section.
- **Layout:** A straightforward approach is to present a **text editor** within the app that loads the rules YAML file content. This way, power users can directly edit the YAML, and it’s the fastest to implement.
- The editor should be a multi-line text area covering most of the window, using a monospaced font. Syntax highlighting is nice but optional (if a third-party code editor component can be easily included, great, otherwise plain text is okay).
- There should be an “Save” or “Apply” button (and maybe “Revert” if they want to discard changes since last save).
- Optionally, a “Validate” button or auto-validation: when clicking Save, the app should parse the YAML and check for errors. If there is a parse error or invalid field, show an alert or inline error message highlighting it. Do not overwrite the old rules file if new content has syntax errors – instead prompt the user to fix.
- If the user tries to close the window with unsaved changes, prompt to save or discard.
- **User Guidance:** Because not all users are comfortable editing YAML, consider adding a short explanation or template in the UI:
- E.g. a label above the text field: “Define your rules below in YAML format. Each rule can specify app_name, title_contains, tag, priority. Example: ...” Possibly even include a small snippet of example in the text area commented out.
- If there's an easier UI idea: We could present a form-based editor (table of rules where each row has fields for app, title, tag, priority). This is user-friendly but more complex to implement. Given "no ambiguity," let's outline the simpler text-based editor for now, as it directly reflects the stored format.
- We might supply a **“Restore Defaults”** or “Reset” button if we ship with default rules (not mentioned, but if we had some defaults).
- Possibly an “Import/Export rules” if needed, but since it’s just a text, user can copy-paste.
- **Editing Features:**
- We could help the user with some conveniences, like auto-completion of app names (perhaps read from the database’s seen app names to suggest) or tag names to avoid typos. This is nice-to-have. For initial version, not required.
- At least consider making the text view vertically resizable or large enough to edit comfortably (maybe 20+ lines visible).
- Use syntax highlighting if possible to color keys vs values or highlight YAML errors (e.g. unmatched indentation). If not, plain text is acceptable.
- **After Save:** When the user saves changes:
- Write the text to `rules.yaml` (backup old one maybe).
- Parse it. If parse fails, show error and do not apply (and maybe revert to last known good version or keep their text for editing).
- If parse succeeds, update the in-memory rules list. Recompute tag list if needed (add new tags, mark unused ones perhaps).
- Optionally ask “Apply to existing data?” as discussed, or automatically apply new tags to at least currently open timeline or future logging.
- We must also update the UI in timeline or stats if the categorization changed for current day: e.g., if user changes the tag name for Slack from “Communication” to “Comm”, the timeline for today should immediately reflect “Comm” tag on Slack entries. This implies we might want to update historical entries’ tag_ids for consistency. However, since we decided not to retroactively alter, those existing ones still point to the old tag maybe. Actually, if we kept the same tag but renamed it, that could be handled by having edited the Tag name in DB rather than adding a new one (an alternative approach: if user changes tag name in rules but same app conditions,

perhaps we detect it's a rename and update Tag record instead of making new. But that requires diffing. Could leave as advanced behavior.)

- Simpler: consider that they might not often rename tags, mostly add new rules or adjust priority. So, on save, just apply going forward.
- Either way, ensure the Stats and Timeline reflect whatever classification is currently stored. If we changed the rules only for new data, then old data remains with old tags (which still exist in Tags table but now no rule produces them – they'll still show in stats until user maybe merges them manually).
- **Alternate UI (if implemented):** If we went for a rule builder UI instead/in addition:
- Could have a list of rules with each rule as a row: columns for App contains, Title contains, -> Tag, Priority. User can add/remove rows. But multi-value conditions make it tricky in a flat table. Possibly have a detail view per rule when selected. Given complexity and timeline, skip this for now.
- Perhaps in future, a nice UI with dropdowns of app names, text fields for title, etc.
- **Testing & Safety:** Provide an obvious way for user to test if a rule works: we might integrate a small test field where they input an app and title and see which tag would apply. Not required but could be helpful in debugging rules. This might be overkill now.
- The rules editor should be only accessible to user and not running in background – just a settings UI. Standard macOS preferences styling (maybe use a NSToolbar with icons for "General", "Rules", "Privacy").

Privacy & Storage Info View

The Privacy view is intended to **reassure the user** about data staying offline and allow configuration of privacy-related settings (like ignore list, encryption, data location).

This likely is another section in Preferences (maybe labeled "Privacy" or "Data & Privacy"). In the Preferences window, user selects Privacy to see this information.

Contents to include:

- **Privacy Statement:** A short text explaining that *"This application operates entirely offline. All data collected (your app usage activity) is stored locally on this Mac and is never sent anywhere."* Emphasize no internet connectivity and no sharing. Also mention that the data is only as accessible as the user's file system (i.e., someone with access to their computer account could technically see it, so encryption option is available if needed).
- **Local Data Storage Path:** Display the location of the SQLite database file (and perhaps the rules file). For example:
- "Data file: `/Users/<User>/Library/Application Support/TimelineTracker/activity.sqlite`" with a button [Reveal in Finder] that opens that folder.
- "Rules file: `/Users/<User>/Library/Application Support/TimelineTracker/rules.yaml`" similarly.
- This transparency allows tech-savvy users to verify the files and perhaps include them in their backups.
- **Data Control:** Provide options to manage the data:
- **Clear Data** button: If the user wants to delete all logged data, we should offer a way. Clicking "Clear All Data" should prompt "Are you sure? This will permanently delete all your activity logs. This action cannot be undone." If confirmed, we can either wipe the database or remove the file and create a new blank one. Also clear tags and markers. The app would effectively start fresh. (This is a good privacy option in case user wants to reset or stop using.)
- Possibly a **Clear Selected Date** or range, but not necessary. Full clear is simplest.

- If implementing, be careful to close DB connections before deleting file, etc.
- **Ignore List Management:** Provide a UI to manage which apps are excluded from tracking:
- Perhaps a sub-section "Ignored Applications". A brief note: "You can exclude apps from being tracked (for privacy or irrelevance). Any time you use an ignored app, it will not appear in your timeline."
- Display the list of currently ignored apps (if none, say “None”).
- Provide an “Add App” button: When clicked, open a file picker or list of running apps to choose an app. Or simpler: show a dropdown of all installed apps (which is hard to get comprehensively). A simple approach: allow user to drag an app into a list or type the app name.
- Possibly integrate with NSOpenPanel restricted to /Applications. The user selects the .app file; we extract its bundle identifier or name to store in ignore list.
- Provide “Remove” button to remove selected app from ignore list.
- Under the hood, store the ignore list in preferences (e.g., an array of bundle IDs).
- Immediately when an app is added to ignore, the logger should stop logging it. If the app is currently active while user adds it, ideally the logger should treat it as if the user went idle (i.e., end the previous session at this moment and not log further until a non-ignored app becomes active).
- Option: If the user had past data from an app they now decide to ignore, we won’t retroactively delete those logs (unless they do clear data or manually remove entries). The ignore list affects future logging.
- Also mention common uses like “It’s recommended to ignore apps like password managers or other apps where even window titles might be sensitive.”
- **Encryption Toggle:** If encryption feature is implemented, provide the control here:
- A checkbox or switch: “Encrypt activity data on disk”. If the user checks it:
    - Prompt for password creation (maybe open a sheet with fields for password, confirm, hint maybe). Indicate that forgetting this password means data becomes unreadable (unless stored in keychain).
    - After user enters password, attempt to encrypt the database:
    - This could involve migrating data to an encrypted database. Possibly show a progress indicator if it’s large.
    - Once done, store the password in keychain (unless we plan to ask every time).
    - Mark in UI that encryption is enabled.
    - If user unchecks (if we allow turning off):
    - Ask for password to decrypt (if not stored).
    - Then write a decrypted copy of DB and replace.
    - Might simplify by only allowing one-way (enable encryption) unless user reinstalls app. But better to allow off too in Preferences with similar flow.
    - The UI should clearly reflect status: either "Data is currently **Encrypted**" or "Data is **Not Encrypted**". The toggle to enable if not, or disable if yes.
    - Also clarify that encryption protects data at rest, but while app is running, data is accessible to the app in memory. For deep privacy, user would quit the app to lock data, etc.
- If encryption fails (lack of library or error), show an error and revert toggle.

- In case we cannot implement now, we might grey out or hide this option with a note "Encryption not available on this system" or so. But spec wise we say it's optional if feasible, so we assume feasible.

- **Permissions Info:** The Privacy section should remind the user about the macOS permissions required:

- For example: “**Accessibility Access:** (Required) The app needs Accessibility permission to read the names of the frontmost app and its windows. You should have been prompted to grant this on first run. If not yet granted, click [Request Permission] to open System Settings > Privacy > Accessibility and enable access for this app.”
- “**Screen Recording:** (Required for window titles) Starting with macOS Catalina, capturing window titles may require enabling Screen Recording permission [2] (even though this app does not record the screen image, macOS categorizes window info as screen content). If you see titles are not being recorded, please grant Screen Recording permission to this app in System Settings > Privacy & Security.”
    - Provide a button [Open Screen Recording Preferences] if possible (Apple doesn’t have an API to jump directly to that pane easily, but we can instruct).
- We should present these in a non-scary way, explaining the app doesn’t actually record video or anything, it just needs the permission to read window names. And that the content is all local.
- If the app detects that required permission is missing, perhaps show a warning icon here or even in the menubar icon (some apps change icon with a caution sign if no permission). At least in Privacy view, highlight if something is missing and guide to fix.

- The app could also have logic on startup to check and prompt if not allowed.

- **Other settings:** Possibly a “Launch at Login” checkbox can go in a General section or here. If not elsewhere, include it in General.

- Actually, we might have a "General" tab for such miscellaneous preferences (like launch at login, hotkey assignment if not covered in marker, maybe UI options).
- The prompt only listed Privacy and Rules explicitly. It's fine to have a General if needed.

- For completeness, mention “Launch at Login” in either General or here:

    - E.g., in Privacy or General: “[ ] Launch automatically at login” (calls SMLoginItemSetEnabled or similar to register helper).
    - “Global Marker Hotkey: ___ [Change]” if letting user change.

- **Window styling:** The Preferences window can be a standard modal preferences window (non-modal actually, just a utility window). Possibly with a tab view or sidebar for sections:

- If using SwiftUI, could use a TabView style .navigationTitle for each section.
- On macOS, typically Preferences is an NSWindow with a toolbar segmented control for each pane. Could mimic that: e.g., toolbar items for General, Rules, Privacy.

- This is mostly aesthetic; any approach is fine as long as sections are separated clearly.

- **Information only:** The Privacy section will mostly display info and have those toggles. There's not much dynamic content except maybe a list for ignore apps. Make sure all text is selectable (so user can copy file path, etc., or copy a help URL if we have one).

- Possibly include a link to a help page or FAQ about privacy if we had documentation (maybe out of scope).

Overall, the Privacy view is about transparency and control: user sees where data is, can wipe it, can secure it, and can limit what's captured (ignored apps). We must ensure that ignoring and encryption features are thoroughly tested as they directly impact logging and data integrity.

All UI elements described should be implemented with SwiftUI if possible, for consistency. SwiftUI's declarative nature will help reflect real-time data changes (like timeline updating, stats recalculating on state change). Use observable objects for the data (e.g., a ViewModel for current day's activities, one for stats, etc.). Navigation between views (Timeline <-> Stats, Preferences panels) should feel smooth and Mac-native.

Because this document is for development, we have enumerated explicit fields, behaviors, and flows to eliminate ambiguity. Developers should have a clear picture of each feature's expectations. Any unspecified standard behavior (window close, etc.) should follow normal macOS conventions.

Storage and Privacy Considerations

This section summarizes how the application handles data storage and privacy, consolidating some points from above and adding any additional safeguards:

- **Local Data Storage:** All user activity data is stored on disk in a **local SQLite database**. By default, this database file will be located in the user's Library under the app's folder, for example:
- `~/Library/Application Support/TimelineTracker/timelog.sqlite` (actual name can be decided in implementation).
- This location is user-specific and not accessible by other users on the system (assuming normal file permissions). It's not in a shared location.
- The database contains the tables described (activities, markers, tags, etc.). No other process should access it, and the file is not transmitted anywhere.
- **Data Retention:** The app will **retain data indefinitely** by default (there is no auto-deletion of old logs). The user has control to clear data if they desire (via the Privacy settings "Clear Data" button). In future, we might allow auto purge older than X days if requested, but not by default.
- **Privacy of Captured Data:** The data captured includes:
- Names of applications you use.
- Titles of windows/documents within those applications.
- Timestamps of when you used them and for how long.
- Idle periods (which indirectly reveals when you were away from the computer).
- Any notes you manually entered.
- **Potential sensitivity:** Window titles can sometimes include personal information (e.g., the title of an email, a document name). We address this by:
    - Storing everything locally only.
    - Providing the **Ignore List** so you can exclude apps that might display sensitive info (e.g. password managers might show your vault name, or maybe your email app if you consider email subjects sensitive).
    - Optionally allowing encryption to protect data at rest.
- The app **does not record keystrokes, content of windows, screenshots, audio, video, or any content beyond those specified strings and times.** It specifically focuses on metadata (app names, window titles, durations).
- **No Data Sharing:** The app sends no data to any server. There is no account system, no cloud sync. Everything is self-contained. This means:
- No telemetry about usage (we don't even collect anonymized usage stats).
- No crash reports sent (unless the user manually sends us something).
- If we included libraries, ensure they also do not phone home.
- Network access is entirely unused. (One can monitor via Little Snitch or similar and see that the app makes zero connections.)
- **Open Source / Inspection:** (If this were an open source app, we might mention code transparency. Not necessary here, but we give user access to their data files at least.)

- **User Permissions (macOS):** To function properly, the user must grant certain permissions:
- **Accessibility Access:** Used to monitor application focus and possibly observe window titles through the AX API. The app will request this on first run by prompting the user (macOS will show a dialog to open System Preferences > Security & Privacy > Accessibility and let user check the app). We must include the usage description in Info.plist (`NSAccessibilityUsageDescription`) explaining why ("Needed to track which app is active for timeline logging").
- **Screen Recording permission:** Due to Apple's privacy design from macOS 10.15+, retrieving window titles (especially from other apps) is classified under "Screen Recording" permission [2], because it could be used to reconstruct information from the screen. Our app is not literally recording the screen, but to get window titles via certain APIs (like CGWindowList) this permission is required.
    - We will include `NSCameraUsageDescription` (for screen) or actually `NSMicrophoneUsageDescription` is audio; specifically for screen recording, Apple has a separate mechanism (it will prompt automatically when CGWindowList is used).
    - In any case, we have to guide the user: "Please enable Screen Recording for [AppName] to allow capturing window titles. This does **not** mean the app will record your screen, only window titles." This is likely a one-time prompt. Without it, the app might only get app names but blank window titles.
    - If user denies, the app should still function (log app name and "(Title not available)" or similar). But we will strongly recommend enabling for full functionality.
- These permissions are solely used locally; the app doesn't send any of the accessible information elsewhere.
- We do not require or ask for other permissions like Contacts, etc., as we don't use them.
- **Encryption:** If the user enables encryption:
- The database file is encrypted with a key derived from the user's password. We might use SQLCipher under the hood which uses 256-bit AES. The key (or password) can be stored in the system Keychain for convenience if the user opts to "Remember Password", otherwise they'll need to enter it each time the app launches to decrypt the data.
- With encryption on, even if someone accesses the raw `.sqlite` file, they cannot read the contents without the password.
- Note: When the app is running and data is unlocked, someone with access to the machine memory or an unlocked session could still theoretically read data (like any running program). But at rest (when computer is off or user logged out), the encrypted file is safe.
- We should warn: if the password is lost and not stored, **data cannot be recovered**. There is no backdoor. The user would have to clear data and start fresh if that happens.
- **Ignore List Effects:** Apps on the ignore list produce no log entries. So if you use an ignored app for 1 hour, the timeline will show a gap for that hour (or it will show as idle if we choose to treat it as idle). Our plan is not to log anything at all, effectively meaning that time will not be accounted for in "Active" time.
- We will *not* explicitly mark it as "Excluded" on timeline because that would ironically reveal that something happened there. For privacy, it will appear as if the user was idle or away from the computer for that period. (Alternatively, we could fill it as idle to keep totals consistent, but then idle is not strictly true either. We lean towards leaving it blank which in practice is the same as idle in that no app was logged.)
- We should clarify this in documentation: "Ignored applications are not recorded at all – time spent in them will simply not show up in your timeline (it will appear as a gap). This is by design to protect that activity from being captured."
- The ignore list itself is stored locally (likely in preferences plist or YAML). We might also treat it as part of privacy config that can be exported. It's not exposed anywhere else.

- **Data Integrity & Backup:** The user's data is important. Although not a cloud service, we should encourage users to include the app's data in their Time Machine backups if they want an archive. We could mention: "Your data is stored in the file above. You may back it up like any other important file. There is no other copy unless you make one."
- **Multi-User:** If multiple user accounts exist on the same Mac, each has their own separate data since it's in their Library. There's no crossover. We do not track if the user is not logged in obviously. If the user Fast User Switch, our app in one account won't see what happens in the other account (and likely gets suspended).
- **No Telemetry:** Reiterating because it's crucial: the app does not even have analytics like "count how many hours user logs" or anything phoning home. If in future we implement update checking, it will be purely checking a version number from a server without sending any user info, and even that could be manual.
- **Compliance:** If we consider GDPR or similar, since we don't share data, most compliance aspects are user's responsibility (their data never leaves device, so no third party processing). We could state that clearly in any documentation.

By implementing these measures, the application should instill confidence in users that they have full control and knowledge of their data. The development team should ensure any code that interfaces with system APIs for logging does not inadvertently create security risks (for example, if using Accessibility API, handle it properly to avoid any potential exploits, and don't log secure input fields even if accessible, etc.). But since we only read window titles and app names, that's fairly safe.

Summary: The product is a privacy-first, local-only time tracker. It collects only necessary data, stores it on-device, gives the user tools to manage that data (ignore, encrypt, delete), and requires user consent for the needed system access. No data leaves the Mac. The developer should verify this by reviewing code paths and ideally by running network sniff tests (to confirm no outbound traffic). This approach differentiates the app from cloud-based trackers and is a key selling point.

End of PRD. The document above specifies all required features, design decisions, and considerations. Developers can proceed to implementation following this blueprint, and any future questions should be resolvable by referring to the above detailed sections (since we aimed to leave no ambiguity).

---

[1] objective c - Best way to know if application is inactive in cocoa mac OSX? - Stack Overflow
https://stackoverflow.com/questions/5039788/best-way-to-know-if-application-is-inactive-in-cocoa-mac-osx

[2] Why do I have to enable screen & audio recording on MacOS?
https://community.startech.com/t/why-do-i-have-to-enable-screen-audio-recording-on-macos/971