ChatGPT

Stage 1 – Precision Search Prompt: Investigate alternative design patterns and frameworks for building a local-first "Codex Session Viewer" that would simplify or improve its architecture and delivery. Focus on approaches that maintain offline functionality and performance (e.g. modular/plugin architectures, using existing IDE or diff-viewer integrations, or adding a lightweight backend) and evaluate their trade-offs versus the current monolithic React+IndexedDB design. Gather evidence of how these patterns improve maintainability, extensibility, and performance in similar tools, and verify that any proposed solution meets key non-functional requirements (fast rendering of large logs, data privacy by offline processing, scalability for more features) given the team's small size and 3–6 month delivery horizon.

EVIDENCE_LOG.md

- **Stage 1 Prompt:** *"Investigate alternative design patterns… versus the current monolithic React+IndexedDB design… verify proposed solution meets key non-functional requirements (fast rendering, offline privacy, scalability)…"* (Stage 1 paragraph above)
- **Current Stack & Scope:** The project is a single-page React 18 + TypeScript app using Vite, Zustand for state, Tailwind UI, and Monaco for diffs, running entirely in-browser with IndexedDB for storage [1] [2]. It implements many features (virtualized timeline, file tree, diff viewer, export, bookmarks, workspace scanning) all in one UI, which increased complexity [3] [4].
- **Monolithic Implementation:** The main `App.tsx` component spans ~1500 lines, importing many sub-components and managing state for file I/O, parsing, UI filters, and diff rendering in one place [5] [6]. This lack of clear separation of concerns contributed to a "jumbled" design. For example, `App.tsx` directly handles file system access via the File System Access API and triggers hashing and diff computation, rather than delegating to distinct modules [7] [8].
- **Design Pain Points:** Running entirely client-side means the browser must handle large JSONL parsing, diff generation, and file scanning. The PRD notes mitigations (streaming parser, virtualized list, web worker for hashing) to handle large inputs [9] [10], but this adds complexity. Browser sandboxing of filesystem requires user permission and complicates scanning logic [11] [12]. The broad feature set risked scope creep, and indeed packaging and a plugin API were deferred to avoid more complexity [13] [14].
- **Comparable Projects:** A similar project "Codexia" chose a **desktop app** approach using Tauri (Rust+React) and even supports a plugin system in its roadmap [15] [16]. Community forks introduced a **Node.js backend with API** to offload tasks from the UI [17], indicating that splitting the architecture can improve performance and maintainability.
- **Potential Improvements:** Established patterns for offline apps (e.g. an extension or plugin-based architecture) could isolate concerns. For instance, building this as a **VS Code extension** would reuse the IDE's diff viewer and file access mechanisms, reducing custom code (with the trade-off of being IDE-specific). A **modular plugin architecture** in the web app itself could allow optional features (like workspace scanning or exports) to be developed and maintained separately, simplifying the core viewer. Standard MVC or state management practices could be applied to better separate the parsing/data layer from UI components, improving code clarity.
- **Non-Functional Criteria:** Any alternative must preserve offline functionality (no server required) and meet performance targets such as fast initial load and smooth scrolling for thousands of events. The current PRD's success metrics demand parse success ≥99%, initial render <1.5s, 60FPS UI on

large sessions, etc. [18] . Thus, proposed solutions are evaluated for impact on performance (e.g. would adding a backend increase load time or complexity?) and on team productivity (small team must be able to implement and maintain it).
- **Evidence of Feasible Change:** The project's own roadmap acknowledged the need for better structure in future (e.g. an installable library distribution and plugin API were planned [13]). This suggests that a more modular, extensible architecture was recognized as beneficial. Leveraging well-supported frameworks (like an Electron/Tauri wrapper for desktop, or reusing VS Code's environment) are standard approaches that can meet the same use-case with potentially less ad-hoc complexity, as seen in community solutions and industry practices [17] [15] .

ARCHITECTURE_REVIEW.md

Architecture Review for Codex Session Viewer

Goals & Constraints

The primary goal of the Codex Session Viewer is to provide a **lightweight, offline tool** to inspect and analyze AI coding session logs (JSONL files) with rich UI features (timeline of events, file diffs, search, bookmarks, etc.) without relying on the Codex runtime or any cloud service [19] [20] . Key constraints include maintaining **privacy and offline capability** (all data stays local), delivering **fast, interactive performance** even on large sessions (thousands of events), and keeping the solution accessible and easy to deploy (ideally as a static web app or embeddable component) [18] [21] . The design must be feasible for a **small team (3–6 developers)** with a limited timeframe, so it should favor well-known technologies and patterns that minimize maintenance overhead. Non-functional targets include:

- **Performance:** Quick load (<1.5s initial render) and smooth UI (≥45–60 FPS on large sessions) [18] .
- **Reliability & Correctness:** High parsing success (≥99% of logs) and robust diff rendering without crashes [18] .
- **Scalability & Extensibility:** Ability to handle growing log sizes and new features (e.g. new event types or collaboration features) without a complete redesign.
- **Security & Privacy:** No data leaks; adhere to least-privilege (e.g. only read user-chosen files/folders) and security of local file access.
- **Operability & Usability:** Easy to install/run (no complex setup or server) and simple to integrate (embeddable or as part of developer workflow).
- **Maintainability:** Clean code structure that a small team can iterate on, with clear boundaries to avoid the "jumbled" all-in-one design that makes adding features risky or slow.

Evaluation Criteria

Based on the goals and non-functional requirements, the evaluation criteria for alternative approaches are:

- **Architectural Simplicity & Clarity:** Does the approach reduce complexity and separate concerns (UI vs data vs integration) better than the current monolith? This improves maintainability for the team.

- **Feature Completeness:** Can it support all core features (log parsing, timeline, diffs, search, bookmarks, exports) either out of the box or via extensions, without significant regressions?
- **Performance Impact:** Will it meet or exceed current performance targets (initial load time, UI responsiveness, memory usage)? For example, offloading work to a backend or worker should improve UI responsiveness, not degrade it.
- **Offline & Privacy Compliance:** The solution must allow offline use and local data processing. Approaches requiring cloud components are disqualified unless they have an offline mode to preserve privacy.
- **Development Effort & Risk:** How much effort for the team to implement and learn the new approach? Are there known pitfalls or uncertainties (new tech risk) that could jeopardize the 3–6 month timeline?
- **Extensibility & Long-Term Viability:** Does it provide a path for future enhancements (e.g. plugin system for new log analyses, or easy packaging into different formats like an IDE extension or npm library)? A superior architecture should make adding features or distribution channels easier, reducing the chance of a future "jumbled" rewrite.

Current Approach Summary

Architecture: The current implementation is a **single-page web application** (SPA) that runs entirely in the browser. It uses React for the UI, with state managed via React hooks and a Zustand store, and bundles with Vite [1]. All data processing happens client-side: the app reads log files (via file input or the browser's File System Access API), parses the JSONL content, and stores session data in IndexedDB for caching [2]. A custom HTML `<codex-session-viewer>` web component wraps the React app for easy embedding on pages [22]. There is no dedicated backend; the only optional server is a dev-time log watcher for telemetry (disabled in production) [23].

Key Features & Modules: The SPA encompasses a timeline view of events with filtering, a file tree and file preview pane, a diff viewer (using Monaco editor for rich diffs), an export functionality (to JSON/Markdown/HTML), and bookmark storage in IndexedDB [24] [25]. It even includes an optional "Workspace Scanner" that can hash and compare the user's project files to highlight differences from the logged changes [26]. These features are tightly integrated. For example, the main `App` component not only renders UI elements but also triggers background tasks (like scanning a selected directory in a web worker and computing diffs) and handles global state like the currently loaded session, filters, and UI modals [27] [28].

Technologies & Decisions: Notable decisions include using **IndexedDB** (via the `idb` library) to persist sessions and heavy data (improving reload speed and offline use) [1], and lazy-loading the Monaco diff editor to avoid large upfront bundle size [29]. The UI uses Tailwind CSS and Headless UI for rapid development of a responsive interface. The choice to implement as a **web component** suggests the intent to reuse or embed the viewer easily in other contexts (e.g. documentation pages or IDE webviews) [22]. The architecture is essentially **monolithic** on the client: while it logically has components (Timeline, FileTree, DiffView, etc.), these are all part of one React app bundle and share state in an ad-hoc way (through React context or passing props). There is limited separation between the data layer and UI – for instance, file parsing and diff computation are invoked from UI components rather than through a well-defined API boundary.

Root Causes of Complexity: Over time, as more features were added, the boundaries in the design became blurred. The app had to handle parsing, data storage, search and filter logic, UI rendering, and

even file system scanning within one process. This led to large components and state management challenges (e.g., the complexity of `App.tsx` maintaining numerous `useState` hooks for filters, selected files, scanning progress, etc. [30] [6]). The **mixing of concerns** (UI with file I/O and data processing) makes the code harder to reason about and test. For example, when loading a session, the UI needs to parse the file (possibly streaming), update state, virtualize the list, etc., all within the React lifecycle, which is error-prone. Additionally, being browser-only imposed constraints: to implement file scanning and diffing, the author had to use the limited browser APIs (with user permission prompts and blob streams) and even create a pseudo-"backend" in a Web Worker for performance [26]. Many of these could have been simpler with a different architecture (like using Node for file access or reusing an existing diff tool). The current design achieved the functionality but at the cost of **high complexity and technical debt**, evidenced by features like a custom file system ignore mechanism in the front-end [31] and multiple fallback code paths to handle performance or permission issues. In summary, the current approach meets the offline requirement and feature goals, but it is **fragile and hard to extend** – adding new capabilities (say, a new export format or supporting "live follow" of a running session) would require careful threading through a tangled codebase, risking regressions.

Option Matrix

We considered several architectural approaches to determine if a better implementation path exists. Below is an option matrix with possible alternatives, including the current approach (for baseline comparison), and their trade-offs:

- **Option 1: Status Quo – Monolithic SPA (Baseline)** – *The current architecture:* a single-page React app, all-in-browser logic.
  Pros: Simple deployment (static files), no external dependencies; data never leaves user's machine (privacy by design) [32]. The team is already familiar with this stack.
  Cons: Poor separation of concerns – UI, data parsing, and storage all intertwined, leading to high complexity. Hard to maintain or extend (risk of one change breaking multiple features). Performance is capped by browser capabilities (e.g., large file handling is limited by memory/CPU of the front-end thread) and complex workarounds (web workers, virtualization) had to be implemented. Scaling up features (or team contributions) will continue to be difficult due to the tightly coupled code.

- **Option 2: Modular SPA with Core Library & Plugins** – *Refactor into a more modular client-side architecture.* In this approach, we keep it a web application but **separate the core logic into a library or distinct modules**, and define clear extension points (a plugin API) for optional features. For example, the "session parsing & data model" could be one module (exposed via a clean API), the "timeline UI" another, and features like exports or workspace scanning could be plugins that listen for events or register themselves with the core.
  Pros: Maintains offline operation and simple deployment (still just a web app), but improves maintainability. A core library could be independently tested and even published (the deferred plan to make an npm package [33]), encouraging reuse. Plugins allow isolating complex or experimental features so that the base viewer remains lean and easier to understand. New event types or functionalities can be added by writing a new plugin rather than modifying core code, reducing risk of regressions. Using an event-driven or plugin pattern (similar to how VS Code extensions work) provides flexibility for future growth (for instance, a plugin could enable collaborative features or live session following when needed, without bloating the core).
  Cons: Initial refactoring effort is non-trivial – the team must design module interfaces and possibly

rework large portions of the code. Over-engineering is a risk if the plugin system is too abstract for a small team. Also, in a pure browser context, plugins would still run in the same thread, so performance gains are limited (though code structure improves). Documentation and governance of plugin interfaces add overhead. However, given that a **plugin API was already envisioned** for future versions [34], this option aligns with long-term plans, just pulled earlier.

- **Option 3: IDE Extension (e.g. VS Code Webview)** – *Reimplement or wrap the viewer as an extension to a popular IDE.* Instead of a standalone app, the logic could live inside an editor like Visual Studio Code. The extension would parse log files (possibly using Node.js APIs in the extension host for efficiency) and present the timeline/diff UI in a webview panel within the IDE.
  Pros: Leverages the existing developer environment: VS Code already has a robust editor, diff viewer, file access APIs, etc. Much of the custom UI (especially diff and file tree) could be replaced or simplified by using built-in components or just showing files in the IDE directly. File system access and performance heavy-lifting can be done with Node (which powers VS Code extensions), eliminating the complex browser-side scanning logic. For users, it's convenient – they can view session logs in the same tool they write code, possibly improving adoption. Development-wise, the team can focus on log parsing and presentation logic, while the IDE handles window management and theming.
  Cons: Ties the tool to a specific platform (VS Code). Users who want a standalone viewer or who use a different editor would be excluded unless separate efforts are made. Distribution through an IDE marketplace is straightforward, but it's no longer just "open an HTML file". Also, the team needs to learn the extension API model (which, while JavaScript/TypeScript, has its own complexity). Debugging in an extension context can be tricky. Performance for large logs should be good with Node available, but the UI is still essentially a browser (Electron) in VS Code, so memory usage should be watched. This option scores high on maintainability (by outsourcing many UI components to the IDE) but low on universal accessibility.

- **Option 4: Desktop Application (Electron or Tauri)** – *Wrap the application in a desktop app container with a native backend.* This approach would create a hybrid architecture: the front-end could remain a React/TS app (possibly the same UI code) but it runs inside Electron (Chromium + Node.js) or Tauri (Rust + WebView) instead of a regular browser. The backend (Node or Rust) can handle filesystem operations, large file parsing, and heavy computations, exposing results to the front-end via an IPC (inter-process communication) or HTTP API.
  Pros: Better separation of concerns and performance: the heavy logic can move to the backend process – e.g., reading and parsing the JSONL, computing diffs using efficient Node/Rust libraries, caching results on disk – leaving the front-end to focus purely on presenting data. This would likely improve performance for large sessions (no more large data parsing in a constrained browser thread) and simplify front-end code (no need for hacks like file handles or web workers for hashing). It remains an offline solution; no internet needed, and data stays local. Packaging for users is straightforward (an installer or binary) and could improve adoption (just run an app). Additionally, frameworks like Tauri emphasize security (Rust backend, minimal binary size) [16]. A desktop app can also open doors for **plugin systems** or extensions in the backend (similar to how VS Code has extensions) because you control the runtime.
  Cons: Higher development and maintenance effort. The team must manage a dual tech stack: web and either Node or Rust. Bugs can arise in the integration between frontend and backend. It's also heavier in terms of distribution (users have to install an app ~ tens of MB). Cross-platform testing (Windows, Mac, Linux) is needed. If not careful, one could end up with a different kind of complexity

– instead of one jumbled JS app, you might get a jumbled JS+Rust app. The timeline to implement this might be longer, though using existing code for the UI softens the blow. Nonetheless, given community interest (there's already a project using Tauri for similar purposes [15]), this is a proven path for more complex tools and could greatly enhance extensibility (e.g. adding new native capabilities in the future, like real-time monitoring).

- **Option 5: Client-Server (Local Web Service + Browser UI)** – *Split the application into a local backend service and a separate front-end.* In this scenario, a lightweight local server (could be Node.js or Python) handles the core logic: it would scan for sessions, parse files, perform diff calculations, and serve the data through an API (HTTP or WebSocket). The front-end would be a web app (could still be React) that queries this service to get session data and displays the UI. The user would run the service on their machine (possibly via a simple CLI), then open the browser UI to interact.
  Pros: Similar to the desktop app approach in separating concerns, but keeps the front-end as pure web (no need for a heavy Electron package). The backend can be optimized for performance and even allow multiple users to connect if needed (for future multi-user or remote sharing scenarios). Development can leverage web skills for front-end and whatever language is best for the backend logic. It maintains privacy (server runs locally). This design might simplify certain features: for example, the backend can pre-index sessions or maintain a database of events for quick search, something hard to do in a purely client app.
  Cons: Increases setup complexity: users have to run a background server process. This two-step usage (start service, then open UI) is less convenient than a one-click app or single-page tool. Ensuring the frontend and backend stay in sync (API versioning) adds maintenance overhead. Also, if the user's environment is restricted (no local Node runtime, etc.), it's a hurdle. From a team perspective, it introduces a need for DevOps considerations (installers or startup scripts, possibly service installation). While this pattern is effective for heavier applications, it might be overkill if the team's needs can be met with a simpler refactor (Option 2) or a packaged app (Option 4) which are more self-contained.

Recommended Path

After evaluating the options against our criteria, the **recommended approach is to refactor the Session Viewer into a modular architecture (Option 2), augmented with selective use of a local backend if needed for heavy operations**. This essentially means starting with the **Modular SPA + Plugin system** as the core strategy, because it directly addresses the maintainability and extensibility issues without abandoning the familiar web technology stack. It keeps the app easy to run (still just open it in a browser or host it statically) and preserves the offline, client-only benefits that were key to the original design [32]. The codebase will be reorganized so that the **core parsing and state management are in a library module** (with a clear API boundary), and features like file-tree workspace sync, export formats, or future enhancements can be implemented as **plugins or separate modules** that interface through event emitters or callbacks.

This recommendation is made because it provides a **clear improvement in modularity with relatively low risk**: the team can leverage existing code (just separating it logically), and no completely new platform is introduced in Phase 1. According to our criteria, it scores high on **Architectural Clarity** (decoupling core logic from UI will shrink that 1500-line `App.tsx` into more manageable pieces, for example), and high on **Extensibility** (a plugin API means future features won't tangle the core). Importantly, it keeps the solution **offline and secure**, exactly as before, satisfying privacy requirements. Performance is expected to **remain**

as good or better: initially it will be similar (since it's the same browser environment), but by isolating components, we can more easily optimize or even swap in web worker modules where appropriate (for instance, the diff computation could become a plugin that runs in a worker thread). This option also leaves the door open to the other options later – for example, once the core is a library, one could relatively easily create a VS Code extension or an Electron wrapper that uses that library. It doesn't preclude those; it just doesn't force them as the first step.

Why not choose the others right now? An IDE-specific extension (Option 3) was tempting due to its maintenance benefits, but it fails the universality criterion – not everyone uses VS Code, and we want the tool to remain a general-purpose viewer. It also adds platform risk (depending on VS Code's API and user base). The desktop app (Option 4) might become a goal in the future (especially if performance needs outgrow the browser), but jumping to it immediately would significantly increase development complexity and timeline risk. It's a larger architectural leap that the small team might struggle to deliver within 3–6 months, whereas a modular refactor is more incremental. Option 5 (client-server split) shares similar concerns – it introduces a lot of operational overhead for uncertain gain, unless the log sizes or collaboration requirements increase drastically. The current scale (single-user, moderate log sizes) doesn't mandate a persistent server process; the overhead outweighs the benefit at this point.

In summary, **modularizing the existing app and implementing a plugin-like extension mechanism is the most balanced path**. It directly targets the root cause of the "jumbled design" – tight coupling – by introducing clear boundaries. It requires effort but is achievable with known technologies. And critically, it positions the project well for future enhancements: once this refactoring is done, the team can more confidently add features or even pursue an Electron or extension version as a Phase 2 with much of the groundwork (the core library) already in place.

Trade-offs, Risks, and Mitigations

Every architectural decision comes with trade-offs. For the recommended path, we acknowledge the following:

- **Trade-off – Immediate Complexity vs. Long-term Simplicity:** In the short term, breaking the app into modules and defining plugin interfaces will add complexity to the codebase (more files, new abstractions). There's a risk of **over-engineering** – e.g., creating a plugin system that is too generic. We mitigate this by **iterating gradually**: first isolate the core logic into a single well-defined module (without fully generic plugins), then abstract one or two features as "plugins" to prove the concept. We will document internal APIs clearly during this process. The long-term payoff is a simpler overall structure despite the initial bump in complexity.

- **Risk – Regression of Features:** Refactoring such a central piece of the app could introduce bugs or performance regressions (e.g., parsing might slow down if not integrated carefully, or some UI filters might break when moving to the new state management). To mitigate this, we plan a comprehensive test suite (see KPIs & Tests) and will do phased rollouts: e.g., release a beta version of the refactored app internally or to power users for feedback before full release. We'll keep the old code path available behind a flag during development as a fallback, so we can do side-by-side comparisons (especially for performance).

- **Risk – Team Learning Curve:** If team members are not familiar with designing plugin architectures or using patterns like event buses, there's a risk of mistakes in the design. We will mitigate by referencing known patterns (for example, how VS Code's extension API is designed, or using a small library for event handling to avoid writing one from scratch). We will also possibly consult an expert or perform design reviews on the module interface before refactoring too far, to ensure the approach is sound (a mini "architecture spike" phase).

- **Trade-off – Not Immediately Tackling Performance with New Tech:** By not jumping to a backend or new platform, we trade away some potential performance gains. Very large logs might still be somewhat slow in a browser. However, the current performance meets requirements, and our plan includes making the architecture extensible – if needed, we can introduce a web worker or WASM module for parsing as a plugin later, or move to Electron in a future phase. The modular refactor lays the groundwork for that pivot without forcing it now. Essentially, we prioritize **reducing complexity** over chasing maximum performance right now, because the latter isn't a pressing issue per the known success metrics (the app already meets 1.5s load target on typical cases) [18] .

- **Open Questions:** A few areas need further clarification as we proceed. One is **how to define the plugin interface** – e.g., will plugins register new UI panels, or just augment data? We will need to decide what the first plugin use-case is (likely the workspace file scanner could be one, or the export feature). Another question is **packaging and distribution**: once modular, do we deliver it as one bundle or multiple? We expect to keep a single bundle for now for simplicity, but in the future an npm package could be published for the core. Additionally, we should monitor whether the refactored app indeed remains within performance budgets; if not, we may need to reassess and possibly accelerate a move to a hybrid (e.g. Electron) approach. Lastly, **user acceptance**: will users notice changes (hopefully only positive, like faster updates or new features)? We might gather feedback once the new version is out to ensure we haven't unintentionally harmed the user experience. These questions will be addressed during the design and prototyping in the migration.

TARGET_ARCHITECTURE.mmd

```
flowchart TB
    User(["User<br/>(Developer)"])
    subgraph Browser Application [**Refactored Session Viewer SPA**]
        direction TB
        UI[**UI Layer**<br/>(React Components)]
        Core[**Core Session Logic**<br/>(Parsing & State API)]
        PluginA[[Export & Reports<br/>Plugin]]
        PluginB[[Workspace Scanner<br/>Plugin]]
        UI ==> Core
        UI --> PluginA
        UI --> PluginB
        Core --> DB[(IndexedDB<br/>Local Cache)]
        PluginB -- file access --> FS[[Local File System]]
    end
    User --> UI
```

```
    %% Legend or Notes:
    %% - UI Layer handles presentation (timeline, diff views) and delegates data
tasks.
    %% - Core Session Logic parses log files, manages session data, and exposes
an API.
    %% - Plugins extend functionality: e.g., Export plugin listens to core data/
events to provide export features; Workspace Scanner plugin interacts with FS
(via browser API or backend) to find file diffs.
    %% - IndexedDB remains for caching sessions and settings.
```

MIGRATION_PLAN.md

Phase 1: Architecture Refactor & Module Separation

Objective: Restructure the codebase to isolate core logic and establish clear module boundaries without altering external behavior.

- **Tasks:**

- *Design Module API:* Define the interface for the core "Session" module (e.g., functions to load a session file, retrieve events, apply filters, etc.). Document this in a short design spec for team review.

- *Extract Core Logic:* Refactor parsing, data structures, and state management out of `App.tsx` into a new module (e.g., `sessionCore.ts` or a `core/` directory). The UI should now call this module to load sessions and subscribe to data updates. Ensure all unit tests for parsing/diff logic still pass using the new module.

- *Integrate Zustand (if not fully utilized):* Use a central store for app state if appropriate (e.g., session data, filter state) that the UI components use via context. This reduces prop drilling and makes the state management clearer after extraction.

- *Modularize Components:* Group related UI components and their helper functions into subfolders (e.g., timeline components, file tree components). This isn't a functional change but improves project structure.

- *Maintain Backward Compatibility:* The public interface (the `<codex-session-viewer>` element usage and any expected behaviors) should remain the same in this phase. Internally, verify that features like open-file, filtering, export still work using the new core module.

- **Gate 1 – Code Review & Tests:** All existing tests must pass with the new structure. Perform a thorough code review focusing on ensuring no feature regressions were introduced. Use sample session files to manually test each feature (open session, filter events, diff view, export, bookmark) in the refactored app. **Exit Criteria:** The refactored app is functionally on par with the old version (or better), with team consensus on the new structure (everyone understands the module boundaries).

- **Timeline:** ~6–8 weeks. *Week 1-2:* Planning and design approval for module interface. *Weeks 3-6:* Implementation of refactoring in small increments (e.g., first move parsing, then state, etc.), with continuous testing. *Weeks 7-8:* Bug fixes, optimization, and code stabilization.

- **Rollback Plan:** Since this is a major refactor, we will do it on a separate git branch. If at Gate 1 the refactored version fails key tests or proves too unstable, we can pause and use the original code on `main` for any urgent fixes. We would then reassess whether to attempt a smaller refactor or seek an alternative approach (e.g., gradually refactor component by component). The existence of the original code path until we merge ensures we can always ship a minor update from the old code if needed.

- **Owners:** Core developer (Lead) – module interface design & parsing logic; Front-end specialist –

component refactoring; QA/dev (everyone) – writing additional tests to cover core functionalities before and after refactor.

Phase 2: Introduce Plugin Architecture & Offload Optional Features

Objective: Extend the modular design by implementing a plugin mechanism and isolate at least one major optional feature as a plugin, validating the extensibility of the new architecture.
- **Tasks:**
- *Design Plugin API:* Define how plugins register with the core or UI. For example, the core could expose an event emitter for session load, or the UI could provide a hook for adding new UI panels. Keep it simple initially (maybe just lifecycle hooks or callbacks). Document this with examples (e.g., "hello world plugin" that logs something) for clarity.
- *Plugin Candidate 1 – Workspace Scanner:* Take the workspace file scanning functionality (which interacts with the File System Access API and does hashing in a worker) and move it into a plugin module. This plugin will be responsible for the "Workspace" panel: it can register a UI component (via plugin API) that the core viewer can show, and handle all its internal logic separately. The core might provide it with session data or events (like when a new session loads, plugin can react).
- *Plugin Candidate 2 – Export/Report:* Similarly, move the export functionality (JSON/MD/HTML export) into a plugin. The plugin API might allow adding new menu items or buttons in the UI. This serves as a second test: one plugin is UI+FS oriented, another is data processing oriented.
- *Testing & Integration:* Ensure that when these plugins are integrated, the end-user experience is unchanged or improved (e.g., the user still sees the file scanning feature and export buttons as before). However, the core app should be able to run **without** these plugins enabled, to prove that they are truly optional. Test the application with plugins disabled vs enabled (simulate by not loading plugin script to ensure it fails gracefully or simply omits that feature).
- *Performance Check:* Measure if splitting into plugins affects load time. It shouldn't significantly, especially if plugins are loaded on demand (e.g., only when user opens "Workspace" feature). If there is a negative impact, consider lazy-loading the plugin code.
- **Gate 2 – Plugin Validation:** We have at least one real plugin working end-to-end. Criteria: The plugin's functionality works as before, and if we turn off the plugin, the core app still runs (just missing that feature). The team reviews the plugin API and is comfortable that it's flexible enough for foreseeable needs (perhaps do a quick brainstorm of another hypothetical plugin to test the API mentally). Unit tests and integration tests should cover plugin-loaded vs not scenarios (for example, test that exporting requires the plugin and fails gracefully if not present). **Exit Criteria:** The app is now plugin-capable, with documentation for how to add a plugin, and two features (scanner, export) implemented that way, with no regressions.
- **Timeline:** ~6 weeks. *Week 1:* Design plugin architecture (could be overlapping late Phase 1 while refactor is fresh). *Weeks 2-4:* Implement plugin system and migrate first feature into plugin. *Weeks 5-6:* Second plugin, testing, documentation updates.
- **Rollback Plan:** If plugin system proves too complex or introduces new bugs that are hard to fix, we can decide to merge the refactored core (from Phase 1) without plugin support into production (since Phase 1 by itself is a net win). The plugin work can continue in a feature branch until stable. This way, the benefits of Phase 1 aren't lost even if Phase 2 takes longer.
- **Owners:** Tech lead – overall plugin architecture design; Developer A – implements Workspace Scanner plugin; Developer B – implements Export plugin; Developer C – focuses on plugin API integration into core and UI points. (Responsibilities can overlap given small team, but assign clear "owners" for each plugin feature).

Phase 3: Optimization & Optional Backend Integration

Objective: Address any remaining performance or scalability issues by possibly introducing a lightweight backend component or further optimizations, and prepare the project for broader distribution.
- **Tasks:**
- *Performance Audit:* Using the KPIs (load time, memory usage, etc.), test the refactored app on large sessions (e.g., 10k+ events, huge diffs). Identify bottlenecks. If parsing large files or generating diffs is still slow, consider implementing those via Web Workers or a WebAssembly module as plugins. (For example, a WASM diff library plugin could replace the JS diff parser for better speed.) If the performance is acceptable, this step may be brief.
- *Prototype Backend (if needed):* If there are clear advantages, prototype a small Node/Rust service for heavy duties. For instance, a Node script that can be invoked to pre-process a .jsonl file into an index or summary. This doesn't mean fully switch to client-server, but provide an optional pathway. This could be a plugin that uses WebSocket to talk to a local service if available. **Only do this if absolutely required** (e.g., for extremely large logs) because it introduces complexity. The outcome might be an **experimental feature flag** where users can opt into using a local helper service for speed.
- *Security & Hardening:* Review the application for any security issues especially around the File System Access API usage and IndexedDB. Implement additional checks or sandboxing in plugins if needed (e.g., ensure the scanner plugin cannot access files outside the chosen directory, etc.). Also run dependency vulnerability scans (npm audit, etc.) and update packages.
- *Documentation & Example Use-Cases:* Update README and docs to describe the new architecture (perhaps an updated diagram, how to add a plugin, etc.). Provide guidance on embedding the viewer as a component or using the core library in other contexts (since we now have that capability).
- *Distribute Beta & Feedback:* Before full release, publish a beta version (maybe on a separate branch or a GitHub Pages link) and allow some users (or internal team) to test it on their logs. Collect feedback on any issues or confusing changes. This will inform final adjustments.
- **Gate 3 – Performance & Security Acceptance:** The refactored, plugin-enabled app should meet or exceed the original performance benchmarks and should pass a basic security audit. Specifically, verify that it still hits parse and render performance targets (use automated tests or profiling). Check that introducing plugins or a backend hasn't opened any obvious vulnerabilities (e.g., no open ports by default, file access stays within user intended scope). **Exit Criteria:** Sign-off from the team that the new version is production-ready: it's stable, fast, and secure. KPIs defined are all green in testing (see KPIs document).
- **Timeline:** ~4–8 weeks (depending on need for backend prototype). If performance is fine, this phase might mostly be documentation and final polish (~4 weeks). If a backend integration is pursued, allocate additional 4 weeks for development and testing.
- **Rollback Plan:** If a risky change (like the optional backend) is attempted and proves too unstable or delays the release, we will omit it from this release. Those experiments can be postponed to a future version. The core product (with plugins) is the priority to ship.
- **Owners:** Performance profiling – assigned to whichever team member has expertise with profiling tools (could be QA or dev); Security review – perhaps involve an external reviewer or use automated tools (all devs contribute fixes); Documentation – technical writer or lead dev; Beta coordination – product manager or lead dev to interface with users.

Phase 4: Release & Monitor

Objective: Deploy the new architecture into production (i.e., release a new version of the application or library) and closely monitor its success criteria, establishing a continuous workflow for future

improvements.
- **Tasks:**
- *Versioning & Release Prep:* Decide on version numbering for this major change (likely v2.0.0 given the magnitude). Prepare release notes highlighting architectural improvements and any user-facing changes. If the project is open-source, communicate with the community about what to expect and encourage plugin contributions.
- *Deployment:* If hosting on a static site (GitHub Pages/Netlify) for the web demo, deploy the new build there. If distributing via npm (e.g., core library or web component), publish the package. For any IDE extensions or optional app wrappers, release those accordingly (this might be beyond this phase if not done yet).
- *Post-release Monitoring:* Although it's offline, monitor indirectly via user feedback. Set up a GitHub issue template or survey for users to report performance or bugs specifically related to the new version. If possible, instrument the app to log performance metrics locally and let advanced users share logs (opt-in) – or simply gather anecdotal reports.
- *Support & Bugfixes:* In the first few weeks post-release, be prepared to address any critical bugs quickly. Have a fast turnaround process for hotfixes if, say, a plugin fails for certain edge-case logs.
- *Measure Success:* After a suitable period (say 4–6 weeks), evaluate the KPIs: Did we reduce the average time to implement a new feature (qualitatively, does the team feel it's easier now)? Are there fewer bug reports related to tangled logic? Performance metrics from test runs – are they consistently within targets? This phase is about closing the feedback loop.
- **Gate 4 – Project Review:** Conduct a post-mortem or retrospective on the migration. Document what went well and any lessons learned. Ensure that all team members are fully up to speed on the new architecture (perhaps a short internal presentation or workshop to solidify knowledge). **Exit Criteria:** The new architecture is stable in production, with positive feedback on maintainability from the team and no significant unresolved issues from users. The project is set up for ongoing iterative development under this new structure.
- **Timeline:** Ongoing/continuous. The release itself is a point in time (targeting end of the 3–6 month window, e.g., month 5 or 6). Monitoring and follow-ups continue as needed.
- **Owners:** Release manager (could be the team lead) – coordinates versioning and announcement; All developers – on rotation for monitoring issues; Project manager – collects metrics and feedback for the review.

Note: Each phase has some overlap potential (e.g., Phase 1 and Phase 2 design work, or Phase 3 testing overlapping with Phase 2 plugin dev). We will use agile iteration cycles to adjust as we learn; the plan is meant to be a guiding structure to minimize risk while making steady progress toward the cleaner architecture.

WORKFLOW.md

Development Workflow for Migration and Beyond

This workflow describes how the team will implement the above migration in a controlled, high-quality manner, and continue with development once the new architecture is in place. It ensures decision gates and quality checks at each step:

1. **Plan & Issue Definition:** All significant work starts by creating a task or GitHub issue (or using our Taskmaster system) describing the feature or refactor to implement. For the migration, we create

specific issues for “Extract Core Module”, “Implement Plugin System”, etc., each with acceptance criteria drawn from this architecture plan.

2. *Decision Gate:* The team lead reviews the plan for the task to confirm it aligns with the architecture goals (e.g., not hacking in a quick fix that violates the new modular design). Only approved tasks move to implementation.
3. **Design Discussion (if needed):** For complex tasks (like designing the plugin API), a brief design is written (could be a markdown in `docs/` or even a comment on the issue). Team members asynchronously review it.
4. *Quality Check:* Ensure the design addresses how the change meets criteria (e.g., “does this preserve offline capability?”, “will this be testable?”). If any design does not meet security or performance guidelines, it’s sent back for revision before coding starts.
5. **Feature Branching:** Developers create a separate git branch (named by feature, e.g., `feature/core-module`) off the latest `main`. All development for that issue happens on the feature branch. For larger efforts, multiple sub-branches may be used (and merged into an integration branch for that phase).
6. **Development & Continuous Integration:** Code is written and frequently committed. Our CI pipeline runs on each push:
7. **Static Analysis:** Linting (ESLint, Prettier) and type-checking must pass. Also, run any security linters (for example, ensure no dangerous APIs are used).
8. **Unit Tests:** All existing tests run on every commit. The developer is expected to add new unit tests for any new module (e.g., tests for core parsing API, tests for plugin interface behaviors).
9. **Build & Bundle:** Ensure the app builds successfully (Vite) and the bundle size is within expected range. CI can warn if bundle grows too large (as a regression check on modularization).
   If any step fails, CI flags it and the developer fixes issues before proceeding. This provides an early gate so that code quality remains high even during refactoring.
10. **Peer Review via Pull Request:** Once the feature branch is at a logical completion (or a milestone for larger phases), a Pull Request (PR) is opened to merge into `main` (or into an integration branch like `plugin-architecture` if not ready for main). The PR description will reference the design and issues, and highlight any trade-offs or decisions made.
11. *Decision Gate:* At least 2 team members review the PR. They check for code cleanliness, adherence to the planned architecture, and adequate test coverage. They also manually test the branch if it's a UI change (pull the branch, run the app, try key flows). For the migration, reviewers will pay attention to whether the new code truly isolates concerns (e.g., no sneaky shortcuts that break the modularity).
12. The PR must meet **Quality Bars**: all CI checks green, no significant new ESLint warnings, and tests added for new functionality. Additionally, we enforce that any public-facing change (even internal API changes) are documented appropriately.
13. If reviewers request changes, the developer addresses them and updates the PR. Only when reviewers approve and all checks pass does the maintainer merge the PR.
14. **Merge and Integration Testing:** Merging into `main` triggers a full CI run and then a deployment of a **testing build** (e.g., to a staging environment or simply a Netlify preview). At this point, the feature is integrated with others. We run a batch of **integration tests** (some may be automated end-to-end tests, e.g., using Playwright to open a sample log in a headless browser and simulate user actions). We also verify that performance metrics are within limits (the CI could run a headless performance test on a known large file and assert timings).
15. *Decision Gate:* If any integration test or performance benchmark fails, the merge is marked for follow-up. We may issue a fix immediately or, if critical, roll back the merge (we can revert the commit) to keep `main` stable.

16. **Periodic Quality Gates:** Given we have multiple phases, at the end of each phase (as described in Migration Plan gates), we do a structured review. This is not just code review but a checkpoint in workflow: e.g., after Phase 1, pause new feature work and focus on fixing any discovered issues, improving documentation, and ensuring all team members are comfortable moving to the next phase. Essentially a mini internal release. Only proceed to Phase 2 tasks when Phase 1 goals are truly met.
17. **Release Preparation:** When ready to cut a release, the workflow includes bumping version, generating release notes, and doing a final full regression test suite. We might use a tag or a release branch. The CI pipeline can be configured to build release artifacts (like a production optimized bundle, or packages) and even run a security scan (SAST/DAST tools, dependency audit). The release is then deployed.
18. **Post-release Monitoring:** Although we can't get runtime logs from user machines directly (no telemetry by design), the workflow includes monitoring indirect signals. We encourage users to report issues; we check discussions or forums. The development workflow accounts for quick patches: if a bug is reported, create a hotfix branch, apply fix, run through CI, reviewers approve, merge and release a point update. Our CI/CD should allow emergency releases outside the normal sprint cadence when needed, while still running all tests for safety.

Throughout this workflow, **automation and quality checks** are emphasized at each gate to prevent regressions. Every code change is tied to an issue (traceability), every merge is validated. This ensures that as we migrate the architecture, we do so without breaking the trust users have in the tool's reliability and without slowing the team down in a quagmire of bugs.

---

workflow.mmd

```
flowchart TD
    subgraph Developer Workflow
        PlanIssue["1. Plan Task/Issue"] --> DesignReview{"Design needed?"}
        DesignReview -- "No (trivial change)" --> branchDev
        DesignReview -- "Yes (write design doc)" --> DesignDoc["2. Design &
Review"]
        DesignDoc --> designGate{"Design OK?"}
        designGate -- "Yes approved" --> branchDev
        designGate -- "No revise" --> DesignDoc
        branchDev["3. Create Feature Branch"] --> codeDev["4. Code & Commit"]
        codeDev --> ciBuild["CI: Lint/TypeCheck<br/>+ Test + Build"]
        ciBuild --> ciPass{"CI Passed?"}
        ciPass -- "No (fix issues)" --> codeDev
        ciPass -- "Yes" --> prReview["5. Open Pull Request"]
        prReview --> peerReview{"Peer Review OK?"}
        peerReview -- "Changes Requested" --> codeDev
        peerReview -- "Approved" --> mergeMain["6. Merge to Main"]
        mergeMain --> testDeploy["CI: Integration Tests<br/>+ Perf Check"]
        testDeploy --> testsPass{"All Tests & KPIs OK?"}
        testsPass -- "No" --> fixBranch["Fix or Rollback"]
        fixBranch --> codeDev
```

```
    testsPass -- "Yes" --> stageGate{"Phase Gate Reached?"}
    stageGate -- "End of Phase" --> phaseReview["7. Phase-End Review & QA"]
    phaseReview --> nextPhase["Proceed to Next Phase"]
    stageGate -- "Otherwise" --> continueDev["Continue Development"]
    nextPhase --> continueDev
    continueDev --> prReview
    nextPhase --> releasePrep["8. Release Prep & Tag"]
    releasePrep --> deployRelease["Deploy Release vX.Y"]
    deployRelease --> monitor["9. Monitor & Feedback"]
    monitor --> bugIssue{"Bug or Feedback?"}
    bugIssue -- "Yes, create issue" --> PlanIssue
end
```

KPIS_AND_TESTS.md

Key Performance Indicators (KPIs)

To objectively measure the success of the new architecture and ensure it meets requirements, we will track the following KPIs:

- **Initial Load Time:** The time to load and display a session (from file selection to timeline rendered). **Target:** $\leq$ 1.5 seconds for a typical session (~MBs of log data) on a developer-grade machine [18]. We will test with sample large logs (e.g., 5k events) to ensure we meet this.
- **UI Responsiveness (Frame Rate):** The timeline scrolling and interactions should remain smooth. **Target:** ~60 FPS (no jank) for up to 5k events loaded, with virtualization ensuring performance [18]. We'll measure frame rate or interaction latency using browser dev tools on large sessions.
- **Memory Footprint:** The app should handle large sessions without excessive memory use or leaks. **Target:** < 500MB RAM usage for a 10k-event session loaded with diffs. (We'll profile memory before and after loading, and ensure proper cleanup when sessions are closed.)
- **Parsing Accuracy:** The percentage of session log files correctly parsed and displayed. **Target:** $\geq$ 99% parse success rate [18]. This is measured by running a suite of real session files (including edge cases) through the parser and verifying no errors or data loss.
- **Export/Function Correctness:** Ensure that outputs (JSON/Markdown/HTML exports) are accurate. **Target:** 100% consistency – for a given filtered view, the export contains exactly those events and correct formatting. This will be verified via automated tests comparing exports to expected files.
- **Extensibility Metric:** (Internal KPI) Time or effort to implement a new feature or plugin. **Target:** e.g., "Implement a new filter or new export format in < 1 week with no major refactor". This is qualitative but we'll assess after migration by attempting a small new feature and see if it can be added cleanly as a plugin/module. A reduction in lines of code touching core areas for new features will indicate success in modularity.
- **Team Velocity and Bug Rate:** Track the number of bugs reported in production related to architecture issues (e.g., race conditions, state bugs). **Target:** a measurable drop in such bugs post-migration (since a clearer structure should reduce logic errors). Also track how many story points or tasks are completed per iteration now vs before – expecting improvement once the team isn't fighting the old design.

Test Plan

We will implement a comprehensive test plan to verify functionality and performance at each stage:

- **Unit Tests:** Continue to expand unit tests for all core logic. For example, tests for the parsing module (feed it various JSONL inputs, assert it outputs the correct event objects, including edge cases for each event type), tests for the diff parser (small diffs, large diffs, binary file detection), tests for utility functions (filtering, hashing). These have already been partly written; we will update them as code is refactored and ensure coverage for new modules. Target unit test coverage is >= 80% of core logic lines.
- **Integration Tests:** Using headless browser testing (e.g., with Playwright or Cypress), simulate user workflows in a real build of the app:
- *Open a session file:* Verify that after choosing a file, the timeline appears with the expected number of events and metadata.
- *Apply filters:* Programmatically click filter buttons or set search text, then verify the UI updates (e.g., filtered count of events) and that only correct events are shown.
- *View diff:* Simulate clicking an "Open diff" button on a FileChange event, then ensure the Diff Viewer shows up with the correct content (we can compare the text in the diff to the known expected diff from the log).
- *Bookmarks:* Mark some events as bookmarked, reload the page (or open the session again), and confirm bookmarks persist (which tests IndexedDB persistence).

- *Export:* Trigger an export (e.g., Markdown export) and capture the downloaded file (Playwright can intercept download). Compare its contents to an expected snapshot for correctness and completeness.
  These integration tests will be run on CI for every release (and ideally for each PR once stable, though they might be too heavy to run on every commit). They ensure that from a user perspective, the app still behaves correctly as we change internals.

- **Performance Tests:** Develop a test script that loads a large dummy session (we have a synthetic generator in the app for 5k events [35]) and measure key timings. We can instrument the app to record timestamps (e.g., time from file load start to timeline rendered event). We'll also use browser performance APIs in tests to capture frame rates during a scroll. If possible, integrate a performance budget into CI – for instance, warn or fail if initial load time > 2s in a headless test environment.
  Additionally, test memory by taking heap snapshots after loading and after closing a session to ensure no large leaks (this might be a manual QA step each release rather than automated).

- **Security Tests:** While offline, we still ensure security through testing:

- Permission tests: Attempt to use the workspace scanner on a restricted directory and ensure the app handles denial gracefully (no crashes, error message shown).
- Data sandboxing: Verify that ignore rules (gitignore, etc.) are honored – we can create a fake file structure and ensure the scanner plugin doesn't pick up ignored files, for instance.
- Dependency audit: run `npm audit` and a tool like DependaBot; the test plan includes a step to update any vulnerable dependency or patch it before release.

- We will also do a static code analysis for common security issues (like not injecting user input unsafely into HTML, etc., though using React largely mitigates XSS issues, and the app doesn't load remote data by design).

- **Acceptance Criteria & User Testing:** For each major feature, define acceptance criteria that must be met before we sign off the migration:

- e.g., *"Given a session with mixed event types, the timeline correctly groups and filters by each type"* – we'll have a test session that covers this and a tester will verify it manually if not automated.
- *"The application should work in Chrome, Firefox, and one other major browser"* – cross-browser testing on final release by manual run-through to ensure no browser-specific issues (particularly with the FS API or IndexedDB).

- We will also recruit a couple of end users (if available) or at least team members not directly coding the feature to do a **user acceptance test (UAT)** of the refactored app using real-world scenarios (load their own logs, use it for an hour, note any issues or performance hiccups). The acceptance threshold is that these users do not encounter any show-stoppers and report the experience to be as good as or better than the previous version.

- **Monitoring & Maintenance:** Since we cannot monitor in production with telemetry, our "tests in the wild" will be user feedback. The plan is to treat any post-release bug as a high-priority test case to add: e.g., if a user log file fails to load, we add that log (an anonymized version if possible) to our test suite to ensure parsing covers it going forward. Similarly, any performance complaints will lead to adding a regression test for that scenario. We will maintain a **KPIs dashboard** internally where after each release or major commit, we log the results of the KPI measurements (e.g., "v2.0: initial load 1.2s for 1k events, 55fps scroll average"), to track trends and catch regressions over time.

Acceptance thresholds: All the above KPIs and tests come together to form the acceptance criteria for the migration. We consider the migration successful and ready for full release when:

- All unit and integration tests pass consistently.
- Performance tests indicate we meet the targets (or have clear justification if a target is slightly missed, with a plan to optimize soon). Specifically, no significant slowdown from the previous version is acceptable – the new version should be equal or better in each measured aspect.
- Security checks show no new vulnerabilities introduced.
- Team sign-off that maintainability is improved (subjectively, code reviews find the new code easier to understand, and objectively, maybe a slight reduction in code complexity metrics or lines of code in `App.tsx` by, say, 50%).
- At least two team members (or beta users) have tested common flows on the new version and did not find major issues. Any minor issues found have been fixed or scheduled.

By adhering to this test plan and monitoring these KPIs, we ensure that the "better way" to implement the project truly results in a superior outcome: a cleaner, more robust Session Viewer that meets its use-case without the previous design's pitfalls.

---

1 2 3 4 9 10 13 14 18 19 20 21 23 26 32 33 34 prd-v3.md
https://github.com/AcidicSoil/codex-session-viewer/blob/078aa28bcbf902538fa54a07d3f8f81c0369ea19/prd-v3.md

5 6 7 8 11 12 27 28 30 35 App.tsx
https://github.com/AcidicSoil/codex-session-viewer/blob/078aa28bcbf902538fa54a07d3f8f81c0369ea19/src/App.tsx

15 16 17 GitHub - milisp/codexia: A powerful GUI/IDE and Toolkit for Codex CLI. Fork chat + FileTree + note system, and more
https://github.com/milisp/codexia

22 24 25 29 31 README.md
https://github.com/AcidicSoil/codex-session-viewer/blob/078aa28bcbf902538fa54a07d3f8f81c0369ea19/README.md