ChatGPT

Codex Session Viewer v3: Technical Specification and Development Plan

Programming Language & Environment

- **Frontend Stack:** The project is built with **TypeScript** (for type safety) and **React 18** (functional components with hooks) using a Vite bundler [1]. We will continue with TypeScript (targeting ES2020+ for modern browser support) and React 18 for all new development.
- **Styling & UI Framework:** The app uses **Tailwind CSS** (upgrading to v4 as available) alongside Headless UI for accessible primitives [1]. We will integrate additional UI component libraries (Shadcn/Aceternity UI, Park UI, Preline) that are compatible with Tailwind. These libraries will be used to enhance the UI with minimal custom CSS, leveraging utility classes and design tokens for consistency.
- **State Management:** Zustand is used for global state (theme, filters, etc.) [1]. We will continue to use Zustand or the existing context for new state (e.g. file timeline modal visibility, active filters), ensuring new components (like a timeline or tag inputs) interact with the central state.
- **Build & Tooling:** The development environment runs on **Node.js** (>= 16.x) and uses Vite for dev and build. No server component is needed (client-only app). We will maintain the same environment, adding any required build steps for new libraries (e.g. Tailwind plugin for Preline, or assets for Monaco).
- **Target Environment:** Modern browsers (Chrome, Edge, Firefox, Safari) supporting ES6 modules and the File System Access API. We will test in Chromium-based browsers primarily (for workspace folder support), with graceful degradation in non-supporting browsers (e.g. workspace features disabled if File System API not available).
- **Distribution:** Continue producing a standalone static web app and an embeddable Web Component (`<codex-session-viewer>`). The build will output `dist/element.js` for easy embedding [2], and we will ensure new dependencies (UI libraries) are tree-shaken to keep this bundle lean.

Code Specifications

Component Architecture & Integration

- **Timeline Component Replacement:** Replace the current custom timeline list with the **Aceternity UI Timeline** component [3]. The new timeline will visually enrich the event display (with sticky headers and scroll indicators) while preserving existing functionality (virtualized rendering, filtering, bookmarks). We will create a wrapper component `TimelineView` that maps session events to the Aceternity UI Timeline data format. Each timeline entry will include timestamp labels (possibly as section headers by date) and event cards as content. The timeline should support rendering rich content (formatted messages, code diffs, images) inline [3]. If Aceternity's Timeline component supports custom content and is performant with large lists, we'll use it directly; otherwise, we'll hybridize it with our virtualization (e.g. using react-virtual to only mount visible timeline items). The timeline entries will incorporate special visual effects for file change events (e.g. highlighting or

"tracking" animations when files are touched) – leveraging the **GitHub Globe** effect from Aceternity UI to visualize file touches over time [4] (for example, pulsing markers on a globe or map of the repository files, if feasible). The new timeline will be implemented behind a feature flag or in a development branch to ensure we can compare performance with the existing implementation and roll back if needed. Existing timeline sub-features (filter chips, bookmarks, copy/export actions) will be re-integrated into the new component's toolbar or as overlay controls.

- **Filter & Search Enhancements:** Introduce a **Tags Input** component for quick filtering of session logs [5]. This will allow users to add multiple filter chips (for event types, file paths, function names, etc.) in a single input field, improving on the current dropdown-based filters. We will use **Park UI's Tags Input** for a robust implementation of tag entry and deletion (built on Ark UI for accessibility) [5]. The tags input's value will sync with our filter state in Zustand. For example, adding a tag "FunctionCall" or "src/App.tsx" will update the corresponding filter criteria and immediately filter the timeline. We will maintain the existing filter logic under the hood (type filter, role filter, text search) but provide this unified UI. Each tag will represent one filter term or condition (we may color-code tags by category, e.g. blue for type, green for file path). The component's API might be `<TagsInput values={filters} onChange={handleFiltersChange} />` where `values` is an array of strings or structured filter objects. **Hover-driven filter menus:** Simplify filter dropdowns such that they open on hover and auto-close on mouse leave (with a short delay) [6]. This removes an extra click for the user; we will use a Headless UI `Popover` or a small custom component to achieve this, ensuring keyboard accessibility is preserved (the menu can still open via focus). Tooltips will be added for all filter icons and buttons using a lightweight library (e.g. Preline or Park UI Tooltip) to improve discoverability [7].

- **File Tree & File Diffs:** Continue using the existing file tree component but enhance it with a **Tree View** UI from Park UI or Preline [8] for better visuals and icons. We will display icons or badges on files that were changed in the session (e.g. a colored dot or file-change icon next to filenames that have diffs) [9]. This will be achieved by analyzing session data for changed file paths (see *Mapping call_ids to file changes* below) and annotating the tree nodes. Clicking a file in the tree currently shows the latest diff preview; we will augment this to show a version timeline if multiple changes occurred. Specifically, a file with multiple modifications will have an expand indicator – clicking it or a context menu option "View history" opens an **Accordion** or carousel listing each change event (diff) chronologically [9]. For the accordion approach, each panel title could be the timestamp of the change (or the event index) and inside is the diff content. This implements the "timeline of file" concept using either a vertical accordion or a horizontal **Stepper/Carousel** UI [10] [11]. We might use Preline's Stepper for a horizontal step-through of file versions (with prev/next controls) [12] or an accordion for vertical stacking (optionally configured via user settings) [13]. The code will treat each file's changes as an array of diffs; we'll provide a function `getFileHistory(sessionEvents, filePath) -> FileDiff[]` that returns all diff patches for that file in chronological order. Each `FileDiff` entry includes at least a reference to the originating event (timestamp or event ID), the diff text, and perhaps the call_id for cross-reference. This function will be used to populate the file history view.

- **Diff Viewer Augmentations:** The Monaco-based diff viewer will remain the primary diff renderer [14], but we will add optimizations and fallbacks for large diffs. We will enforce a threshold (e.g. ~1MB or X lines) above which we do not auto-render the diff in Monaco (to avoid freezing) [15]. Instead, for oversize diffs, we'll show a simplified view: possibly just the first few hundred lines with a

message “Diff too large to display fully. Download full diff?” and a download link. This logic will live in our `DiffView` component: e.g. `if(diff.size > LARGE_DIFF_THRESHOLD) { renderLargeDiffFallback(diff) } else { renderMonacoDiff(diff) }`. For binary files or diffs that appear to be non-text (we can detect by binary characters or file extension), we will show a message “Binary file – cannot display diff” [16] . We will also integrate a **Canvas reveal effect** from Aceternity UI on diff cards [17] – this could be a subtle animation when a diff card comes into view, visually highlighting the code changes. The diff viewer already supports split vs inline view and word wrap toggles [16] ; we will ensure these continue to work after our modifications. We will also evaluate alternative diff rendering libraries (like `diff2html` or CodeMirror with diff gutter) for performance with huge diffs [15] – if one proves significantly faster for very large content, we might integrate it as an optional renderer for those cases (selected dynamically).

- **Session List Datatable:** The “All Sessions” modal (currently a simple list with search) will be upgraded to a paginated **Data Table** for better usability [18] . We’ll use Preline’s datatable component for a pre-built responsive table UI [18] . The table will display session metadata columns (Session ID, date, number of events, etc.) with sorting and filtering capabilities. We will fetch the session list from our IndexedDB `sessions` store (via `listSessions()` which returns saved session metadata [19]). The code will populate an array of rows for the datatable, and we’ll plug it into the Preline component initialization (ensuring to include the Preline JS plugin for datatable functionality if required). Each row will also have an action (e.g. a “Load” button or clickable row) to open that session. The datatable will support searching by session name or ID (client-side filtering since N is likely small). If the number of sessions is very large, we’ll implement virtualization or pagination as needed (Preline might provide pagination out-of-box).

- **Enhanced UI Effects:** Implement various UI flourish components from Aceternity UI to improve the look and feel:

- **Sparkles Effect** under the main title [20] : Use the Sparkles component to add a subtle sparkle animation behind or around the app title (e.g. “Codex Session Viewer”), giving a dynamic feel. This will likely involve wrapping the title in `<Sparkles>Title</Sparkles>` if using a provided component, or injecting a canvas overlay with the effect. We will ensure this effect is purely decorative and does not hinder readability (respecting prefers-reduced-motion to disable if user prefers).
- **Background Beams** for the app background [21] : Introduce a background canvas or SVG effect with beams of light or color, via the Background Beams component. This will run once (on load or idle) to avoid continuous heavy animations. We’ll integrate it as a component overlay behind the main content container.
- **Vortex Animation** [22] : This could be used on specific elements (maybe the welcome screen or behind the session metadata panel) to draw attention. We will use it sparingly (e.g. on hover of the “drop file” area or as a background of an empty state) to not distract from content.
- **GitHub Globe** effect [4] : As noted, we intend to use this to visualize file changes. If not literally the globe, possibly a network graph or rotating globe showing points for each file changed. The integration would be to feed the component a set of “points” (file changes) when a session loads or on demand. This might be an optional visualization toggle in the UI (e.g. a button “Show Repo Globe” that opens a modal or side panel with the globe).

- **Canvas Reveal** on diffs or images [17] : Use this effect so that when a user expands a diff or message card, the content appears via a canvas shader effect (e.g. wipe or fade in). We will integrate by wrapping diff content in the provided `<CanvasReveal>` component if available. These enhancements are mostly visual; code-wise, they involve importing the component from the library and placing it appropriately in JSX, plus potential initialization calls. We'll ensure all these effects are isolated (e.g. not interfering with each other, and can be toggled off for performance troubleshooting if needed).

- **Message View Options:** Add a side-by-side chat view mode for user/assistant messages [23] . Currently, messages appear in the timeline vertically. We will introduce an alternate rendering for `Message` type events: a component `ChatPair` that can place a user message and the following assistant message side by side (if they are sequential in the log). This could be a toggleable view mode: when enabled, the timeline groups consecutive user+assistant messages into a single card with two columns (user on left, assistant on right). This uses a design similar to a chat transcript. The implementation uses CSS grid or flexbox for the two columns. We'll ensure long content scrolls or wraps appropriately. This is an **optional UI** setting in the viewer's settings panel (for users who prefer reading dialog in chat format). The grouping logic will detect pairs by event type and role (we have role info on Message events [24]). We'll keep the default as linear for now and allow side-by-side as an enhancement mode.

- **Modularization & Reusability:** All new UI components (timeline, file carousel, filters bar, etc.) will be implemented as separate React components or hooks within the project's structure (`src/components/...`). We will follow the existing project conventions for file organization. For instance, new UI elements from shadcn/Aceternity UI might reside in `src/components/ui/` directory, possibly with their own subfolder if they come with multiple files (e.g. `timeline.tsx` , `sparkles.jsx` etc.). We'll isolate styling concerns to those components (using Tailwind classes or library-provided styles) so they can be reused or updated independently. We will also document their usage within the code (comments or a README section) to ease future reuse, especially with the goal of potentially packaging this as a library.

Data Handling & Logic

- **Session Parsing:** Continue using the streaming JSONL parser with Zod validations to read session files [25] [26] . The parsing function (e.g. `parseSession(file: File)` or similar) will output a `ParsedSession` object with `meta: SessionMeta` and `events: ResponseItem[]` [27] [28] . We will extend the `SessionMeta` or events structure if needed to accommodate new features. For example, if `SessionMeta.git` info is present (branch, commit) we ensure it's parsed (already part of the schema as per GitInfo) and perhaps use it to label the timeline start or file versions. If new event types like `FileChange` are introduced in sessions (noted as supported in PRD [29]), we will update the parser and UI to handle them (possibly treating them similarly to our derived file diffs). The parser's error tolerance remains important – any unknown or schema-drifting events should be logged and shown in a generic way rather than breaking the app [26] . Input: a File object or file content stream; Output: `ParsedSession` or an error object with parse errors and the partial data.

- **Mapping Call IDs to File Changes:** Implement logic to correlate `apply_patch` function calls with the files they add or modify, using the unique `call_id` present in those events [30] . We will create

a utility (e.g. `analyzeFileChanges(events: ResponseItem[]) -> Map<string, FileChange[]>`) that scans all events for patterns indicating file edits:

- It will find all `FunctionCall` events where `payload.name === "shell"` and the `arguments` contain `"applypatch"` (or look for the `"*** Begin Patch"` marker). Each such event has a `call_id`.
- It will then find the corresponding `function_call_output` event with the same `call_id`, which typically contains an output string listing files that were added/updated (e.g. "Success. Updated the following files: A src/...") [31]. From this, we can extract the file paths and whether they were added (A) or modified (M).
- We will also parse the patch content itself for file path headers (`*** Add File: path` or `*** Update File: path`) to double-check we catch all files, especially if multiple files are patched in one call. (Codex often patches one file per call, but it could patch several in one go – our code will handle either case.)
- For each file path changed, we create or append to a `FileChange` entry that includes the `call_id`, the diff text (from the patch), and a timestamp. These could be stored in a structure like `fileChangesMap[filePath].push({ callId, diff, timestamp })`.
- Additionally, we maintain a `callIdMap[call_id] = { files: [file1, file2,…], outputEventIndex, success: boolean }` for quick lookup if needed (e.g. linking an event card to the files it changed).
- This mapping will be run after parsing a session (or on-demand when a session is loaded) and cached, since it's derived data. It enables multiple features: marking files in the tree that have changes, listing all files changed by a given call (e.g. on an event card we could list "Files changed: ..."), and retrieving file history (by looking up all call_ids that affected a given file).
- **Data Structures:** We will likely introduce a new TypeScript interface to represent a file change instance, e.g.

```
interface FileChangeEvent {
  filePath: string;
  callId: string;
  timestamp: string;
  diff: string;
  action: 'added' | 'modified';
}
```

and then use `Record<string, FileChangeEvent[]>` for filePath to changes mapping, and `Record<string, string[]>` for callId to filePaths mapping. These can be attached to the ParsedSession (e.g. `ParsedSession.fileChanges`).

- **UI Integration:** The timeline will use this info to perhaps label events: for each FunctionCall event that added/updated files, append a visual cue (like an icon or color stripe) indicating it's a code change event. Clicking that icon could trigger a focus on the file in the file panel. The file tree uses the map to highlight changed files as described. The file history modal uses the chronological list of diffs per file from this map. We will ensure the mapping is correct by using the real session examples (from logs) to test it (as shown in ROADMAP examples, each add/update call with call_id has a matching output) [30] [32]. If an event is missing an output (e.g. if a patch failed and no output event

came, or output not captured), we handle gracefully by using whatever info is available (partial diff or just marking the file as changed without diff content).

- **Function & Component Signatures:** Key functions will be updated or added:

- `loadSession(file: File | string): Promise<ParsedSession>` – already exists to parse and load a session. We will ensure it invokes the new mapping logic after parsing. If loading from IndexedDB by id, it should retrieve stored events and similarly run mapping (or store mapping results in the DB for quick reload).
- `filterEvents(events: ResponseItem[], criteria): ResponseItem[]` – extend this to handle new filter types from tags input. For example, if criteria includes file path filters or function name filters, apply those. The signature might become `filterEvents(events, { text, types, roles, filePaths, fnNames, ... })`. We'll include logic so that if a file path filter is set, we check each event: if it's a function call or output that touched that file (we can check via callId->file map) or if it's a `Message` that mentions that path (less likely), etc. This ensures the tags input that filters by file name will narrow down to relevant events.
- `exportSession(format, options)` – if needed to incorporate any new data (likely unchanged, but ensure that if "bookmarks only" or filters are on, it still works [33]). We might include file-change context in exports in the future, but not explicitly requested now.

- UI component props:

  - `TimelineView props: { events: ResponseItem[], groupBy?: 'none' | 'call', filters: FiltersState, onEventSelect?: (event) => void, ... }`. If `groupBy='call'`, TimelineView will group function call & output together (e.g. render as one timeline item with expandable details – possibly using the Accordion for grouping events per call) [13]. We will implement grouping as an enhancement: e.g. group function_call with its function_call_output and reasoning if any, to declutter the timeline. The default will be `groupBy='none'` (each event separate), unless a user setting toggles grouping.
  - `FileTree props: { files: FileNode[], changedFiles: Set<string>, onFileSelect: (path) => void }`. We'll extend `FileNode` to hold an indicator if changed (or use `changedFiles` set to mark them). The component will render with icons as described. The `onFileSelect` will open the preview or diff as now.
  - `FileHistoryModal props: { filePath: string, changes: FileChangeEvent[], onClose: () => void }`. This is the new modal/accordion that shows a sequence of diffs for one file. It will map over `changes` array to produce either a stepper or accordion UI. Each diff can be shown either as a mini unified diff or with an "Open Full Diff" button to launch Monaco for that version. We will include navigation controls if using a carousel/stepper (prev/next arrows).
  - `TagsFilterBar props: { filters: FiltersState, onFiltersChange: (FiltersState) => void }`. This wraps the tags input and possibly existing filter dropdowns if any remain (like role filter might still be a separate toggle since that is secondary to Message type).
  - `Toast props: { message: string, type: 'info'|'warn'|'success', ... }`. We'll implement a lightweight Toast system (using Park UI's Toast or a minimal custom) to show ephemeral notifications. For example, after loading a session, if our scan found X file

changes or Y potential issues, we can notify the user ("Loaded session with 5 file changes" or "Workspace synced – 2 files differ from session"). This addresses the note about showing matches on load [34]. The toast component likely doesn't need many props beyond message and maybe auto-dismiss timeout; we will ensure it can be triggered from anywhere (perhaps via Zustand or a context).

- **Migration Plan:** We will introduce these enhancements in a backward-compatible way wherever possible:

- The existing timeline code (likely a list with virtualization) will be retained initially, but behind a prop or flag we can switch to the new TimelineView. We will thoroughly test the new timeline with large sessions (thousands of events) to ensure performance is acceptable. Only then do we remove the old timeline implementation. The data format for events remains the same, so most business logic (parsing, filtering) stays in place – we're mainly swapping the rendering layer.
- The diff viewer remains the same component; we'll just add checks within it (so no separate migration needed, just additional code paths for large diffs).
- Filters: replacing the filter UI with tags input is a significant UI change, but the underlying filtering mechanism is the same. We'll keep the old filter UI code until the new one is stable. We might hide the old filter bar via feature flag and show the new tags input bar, ensuring all filter options are representable (for role filter, perhaps the tag input could spawn tags like "role:assistant"). If needed, we keep a simple dropdown for role filter if that's easier.
- The session list modal turning into a datatable is straightforward – we can replace it outright since it's not heavily integrated elsewhere. We ensure the new table covers all old functionality (search by name, listing file count maybe).
- The file tree changes (icons, history modal) augment the existing tree. We will implement the new FileHistoryModal and only invoke it when a user explicitly requests to see file history (so it doesn't affect anything if unused). The tree itself just gets extra icon indicators, which is non-breaking.
- The new UI libraries (Aceternity, Park UI) will be incrementally added. We need to be cautious about style conflicts: for example, Park UI uses Panda CSS (a CSS-in-JS) – instead of fully integrating Panda, we might extract just the needed styles or adapt the components to Tailwind. The migration plan could involve initially copying the needed components into our codebase (Shadcn-style, where we have full control over them) and tailoring the styling to match our Tailwind setup. This ensures we're not locked into an external style system and can maintain a cohesive design. We will document any such adaptation.
- We will remove any deprecated code once the new components are in place. For instance, the old filter dropdowns code can be removed after the tags input is thoroughly tested, and the old timeline JSX will be removed once the new one proves itself. All removal will be done in major version bumps to signal breaking UI changes, though if we maintain the web component API, we should ensure the outside interface is the same (the internal changes should not affect how a host page includes or interacts with `<codex-session-viewer>`).

Mapping Session Data to UI (Call IDs, Events, and Diffs)

A crucial integration is correlating session events with file diffs and reflecting that in the UI: - Each file addition or update in the session is associated with a unique `call_id` [30]. We will use this fact to link the *function call event* that initiated a change and the *file(s) it changed*. In practice, whenever the timeline renders an event that changed files, it can now also present quick links or context about those files. For example, a FunctionCall card might have a list of filenames (as badges) that were added/modified by that

call. Clicking a filename badge could open the diff directly or highlight it in the file tree. - The timeline grouping feature (Accordion grouping by call) will use call IDs to combine a function call with its result output and any related sub-events. This means instead of two separate entries for a call and its output, we show one expandable entry. The grouping function will produce a data structure like `{ callEvent, outputEvent }` for each call_id, so the TimelineView can render them together. We will ensure that bookmarking still works in grouped mode (if a user bookmarks either the call or its output, the group could be marked). - The file history feature essentially inverts this mapping: given a file, find all call_ids that affected it, then retrieve those events. We will sort those events by timestamp to reconstruct the file's change timeline. This allows a user to step through how a particular file evolved during the session, addressing the roadmap note on viewing file versions with timestamps [11]. The underlying mapping function described earlier provides this easily (by looking up file in fileChangesMap). The UI just needs to present it cleanly (e.g. "Version 1 (added at 10:33:10Z by call X)", "Version 2 (modified at 10:34:44Z by call Y)", etc., with diffs). - We will maintain these mappings in memory (or IndexedDB if we decide to cache for very large sessions) for quick access during user interactions. The mappings will also allow potential future features like "jump to next change of this file" in the timeline or "diff against workspace version" if a workspace is connected.

Input/Output Parameters for Key Functions

To clarify the interfaces of important functions and components, here are the expected inputs and outputs:

- `parseSession(file: File | string)`:
  Input: A File object (from file picker or drag-drop) or a file path string (if loading from a known location).
  Output: A `Promise<ParsedSession>` resolving to an object containing session metadata and an array of event objects. For example:

```
interface ParsedSession {
  meta: SessionMeta;
  events: ResponseItem[];
}
```

  where `SessionMeta` includes `id`, `timestamp`, and optional info like instructions or git info [27]. `ResponseItem` is a discriminated union of event types (Message, FunctionCall, FunctionCallOutput, LocalShellCall, WebSearchCall, etc.) [35]. If parsing fails, the function may throw an error or return a special `ParsedSession` with `events` empty and an error message. We validate each line with zod, but unrecognized lines are still included as type `Other` for completeness [26].

- `analyzeFileChanges(events: ResponseItem[])`:
  Input: The array of events from a session.
  Output: An object with at least two mappings:

```
{
  fileToChanges: Record<string, FileChangeEvent[]>,
```

```
  callToFiles: Record<string, string[]>
}
```

`fileToChanges` maps a file path to all changes (add/update) in chronological order. Each `FileChangeEvent` includes `callId`, `timestamp`, `action` (Add/Update), and possibly the diff or patch snippet. `callToFiles` maps each call_id to the list of file paths it affected. For example, `callToFiles["call_123ABC"] = ["src/App.tsx", "src/utils/helpers.ts"]` if one patch touched two files.
This function does not modify global state; it purely derives data. It will be called after parsing (and whenever we load a new session) and its result stored (in memory or state). It leverages patterns like the `"*** Add File:"` or `"*** Update File:"` markers in patch text and the presence of matching output events indicating success [32].
If `events` is empty or has no file changes, it returns an empty mapping. Performance: O(N) over events, with string searches in patch text – acceptable for sessions of a few thousand events. We will write unit tests using known log excerpts to verify that `analyzeFileChanges` correctly identifies all changed files (per the ROADMAP examples) and that each call_id maps to expected files.

- `filterEvents(events: ResponseItem[], filters: FiltersState)`:
  Input: A list of events and a filters object containing criteria (could be shape like `{ text?: string, types?: Set<MsgType>, roles?: Set<Role>, filePaths?: Set<string>, fnNames?: Set<string>, onlyBookmarks?: boolean }`).
  Output: A filtered array of events that match all active criteria.
  The function checks each event:

  - Text filter: if provided, event content (message text, function arguments, etc.) should contain the substring (case-insensitive).
  - Type filter: event's type (e.g. "Message", "FunctionCall") must be in the allowed set [36]. If type filter includes the pseudo-type "ToolCalls", that means allow FunctionCall, LocalShellCall, WebSearchCall collectively [37] – our logic will handle that grouping.
  - Role filter: if active (only relevant when type includes Message), check event.payload.role equals the selected role [24].
  - FilePaths filter: if provided, include events that changed those files or reference them. We use `callToFiles` mapping here – if an event has a call_id that is in the map and any of the files overlap the filter set, we include it. Also, we might consider FileChange events if present as separate event types.
  - Function Names filter: similar approach – include events where (if FunctionCall type) the function name is in the set. E.g. filter for "shell" or "webSearch" calls.

  - Bookmarks: if `onlyBookmarks` is true, only include events that are bookmarked (we have a list or set of bookmarked event IDs in state).
    The result array will be used to render the timeline. This function will be invoked whenever filters change. Complexity is O(N * number_of_filters), which is fine for client-side given N~several thousand max. We will keep it optimized (e.g. skip filePaths loop if no filePaths filter set).
    Example: filters = { types: ["FunctionCall"], filePaths: ["src/App.tsx"] } would yield only function call events that touched App.tsx.

- `exportSession(format: "json"|"markdown"|"csv"|"html", options)`:
  Input: A format string and optional parameters (like whether to include only filtered events or all, which columns for CSV, etc.).
  Output: A blob or string for the exported content, or triggers a download.
  This function likely already exists; we'll ensure its interface remains the same. It uses the currently filtered events (from state) to produce output [38] [39]. We might extend `options` to include context like session name for filename, etc., but mostly it stays consistent. Error handling: if an unsupported format is requested, it should throw or default to JSON. After generation, it returns a Blob or data URL which the UI uses to prompt download. (For example, `exportSession("markdown")` returns a Markdown string or triggers a file download named like `${sessionId}.md` as per current behavior [33].)

- **Component Props (Input/Output):**
  For React components, the input is props and output is rendered UI or callback events:

- `<codex-session-viewer>` **Web Component**: It encapsulates the entire app. It can accept attributes or methods for loading data (e.g. a `loadSession(id or file)` method). The output is just the embedded UI. We will keep the external API unchanged (so existing integrations don't break), only expanding internal functionality. If new external methods are needed (for example, a host page might want to programmatically apply a filter or get a list of changed files), we will consider adding methods like `getChangedFiles(sessionId)` or `setFilters(filters)` to the custom element's class, with careful documentation.
- **Event callbacks**: Some components output user interactions via callbacks, e.g., when a user selects a session from the list or clicks a timeline event. We will either handle these internally (since it's mostly a self-contained app) or expose certain callbacks if needed for integration. For instance, if `<codex-session-viewer>` had a prop for `onEventSelected`, we might fire an event when an event is clicked. However, as a spec for development, the focus is on internal I/O, so we ensure functions like `onFileSelect(path)` for the FileTree triggers showing that file's diff, `onEventSelect(eventId)` in the Timeline triggers maybe focusing or highlighting that event's card, etc., are properly wired.

Error Handling & Validation

Robust error handling will be implemented across the application to ensure a smooth UX even when unexpected inputs or failures occur:

- **Session File Parsing Errors:** If a loaded session file is malformed or contains invalid JSON lines, the parser will catch Zod validation errors [26]. Instead of crashing, we'll log the error to console and show the user a message in the UI. For example, an error banner in the timeline area: "⚠ Some entries in the session file could not be parsed and were skipped." The app will still display whatever events were parsed successfully. If the file is completely unreadable (e.g. not JSONL), we'll show a modal or toast error: "Failed to open session – unrecognized format." The UI will return to the welcome screen state safely.
  Additionally, if a session's meta line is missing or corrupt, we'll generate a placeholder SessionMeta (id "Unknown") so that the rest of the file can still be treated as events. This follows the tolerant parsing approach [26].

- **File System Access Errors:** When using the optional workspace folder connect feature, handle permission denials gracefully [40]. If the user cancels the directory picker or the browser refuses access, we'll simply disable workspace diff features and perhaps show a tooltip or toast: "Workspace not connected. Some file comparison features are unavailable." If an IndexedDB operation fails (e.g. the user's storage is cleared or blocked), catch it and show a notification. For example, if saving a session to the library fails, we warn "Could not save session state for reopening. Make sure storage is enabled." All such errors should not prevent core usage (the user can still view the session live).

- **Diff Rendering Guardrails:** For very large diffs or binary files, as mentioned, we do not attempt to fully render them. Instead, we present the user with an informative message [15]. This is both a performance and error-handling concern – it prevents the app from hanging or crashing due to out-of-memory. We will implement try/catch around diff parsing as well (if using a diff parser library or Monaco's createModel, which might throw on huge input). On catch, we fallback to the message "Diff is too large to display" with an option to download it as a file. This way, even if something unexpected happens in diff generation, the UI remains stable.

- **Network/Resource Loading Errors:** The app is largely self-contained, but it lazy-loads the Monaco editor and possibly other chunks. We will handle module loading failures (e.g. if Monaco fails to load due to network issues, or the user is offline but tries to open a diff). In such cases, our Diff Viewer already has a plain text fallback [41]: we'll ensure it's robust. If Monaco import fails (Promise rejects), we catch it and set a state like `monacoAvailable=false`, then use a simple `<textarea>` or `<pre>` to show the raw diff text as a fallback, with a message "Plain diff view (editor unavailable)". We will also log this for debugging.

- **Validation of User Inputs:** There are not many free-form user inputs (no forms to submit), but wherever the user provides input (e.g. filtering text, tag inputs, file path in two-file diff tool), we will validate and sanitize. For instance, the two-file diff expects two file paths from the workspace; we will ensure the selected files exist and are text-based (maybe check file size/type, and catch errors reading them). If a user manually enters a path that cannot be read, we'll show an error under the input. For tags input, if we later allow custom tag categories (like `type:FunctionCall` shorthand), we will parse and ignore tags that don't match known filters, perhaps highlighting them in error (or simply not adding). All user text displayed (like in search highlights) will be properly escaped to prevent any injection issues in the UI.

- **Fallback UI for Missing Features:** If a UI library component fails to initialize or is not supported in the browser, we will fallback. For example, if the Aceternity UI Timeline is not yet available or has a runtime error, we will detect that (maybe via try/catch around its usage) and revert to a simpler list. If a fancy effect (sparkles, globe, etc.) cannot load (say a 3D context error), we catch it and perhaps disable that effect with a console warning. The presence of these effects is purely enhancement, so failure should not impact core functionality. We might also allow the user to disable effects in a settings menu if they experience issues (e.g. a "Toggle visual effects" switch that removes beams, sparkles, etc., which also is a form of error mitigation for low-powered devices).

- **Graceful Handling of No Data/Empty States:** In any view that could be empty (timeline after filters, file tree if no files, search no results, etc.), provide a friendly message rather than blank space. For example, if the timeline filter yields no events, show a centered note "No events match the current

filters." If the file tree has no entries (e.g. session with no file changes and no project files), show a placeholder "No files to display." These are not errors per se, but prevent user confusion.

- **Toast Notifications for Key Errors:** Use the planned Toast system to surface non-blocking errors or important info. For instance, after loading a session, if we detected that some events were skipped due to parse errors, we can fire a toast: "⚠ Loaded with some errors. Some events could not be parsed." Or if the user tries to open a second session without saving the first, maybe a toast "Opened new session (the previous session remains in your library)." Using toast keeps the user informed in context without modals that require dismissal.

- **Logging and Debugging Aids:** We will keep dev logging hooks (the project mentions optional local dev log server) [42]. For production, if an unexpected error happens (like an unhandled exception in a component), we will catch it via React Error Boundaries (the app already has `<ErrorBoundary>` in places [43] [44]). We'll extend ErrorBoundary usage around new components (timeline, file history) to capture any errors there and show a fallback UI. For example, if the Timeline component throws, the boundary can show "Timeline failed to load. Try refreshing the page or check the console." This ensures the app doesn't fully white-screen on any one component error.

In summary, the strategy is to **never let the UI silently break**. Every potential failure point has a fallback path or error message, ensuring the user can continue using other features. All error messages will be phrased in user-friendly terms (no raw stack traces or technical jargon) and, when appropriate, guide the user (e.g. "please try a smaller file" or "enable storage permissions to use this feature").

Code Quality Standards

We will uphold high code quality standards throughout the implementation:

- **Consistent Style & Conventions:** Follow the existing naming conventions of the project for all new code. React components will be PascalCase (e.g. `TimelineView`, `FileHistoryModal`), while variables and functions are camelCase (`analyzeFileChanges`, `getFileHistory`). We'll use clear, descriptive names – for instance, prefer `filteredEvents` over `events2`. All code will be formatted with Prettier (ensuring the project's Prettier config, if any, is applied) to maintain consistency in spacing, quotes, etc., across the codebase.

- **Type Safety and Clarity:** Leverage TypeScript to its fullest – define interfaces for complex data (SessionMeta, ResponseItem, FileChangeEvent, etc.) and use them in function signatures. Avoid using `any`. If any part of the code is uncertain (e.g. a JSON structure that might evolve), use appropriate union types or generics rather than disabling type checks. We will also update or add JSDoc/TSDoc comments for all major functions and components, describing their purpose, params, and return values. For example, above each utility function like `analyzeFileChanges`, include a brief description and an `@example` usage snippet. This documentation not only helps future maintainers but can be used to generate reference docs if needed.

- **In-line Documentation:** Important or non-obvious logic will be commented. For instance, in the filter function, we might add a comment block explaining the filtering combination logic. In the

timeline grouping code, a note about how call events and outputs are paired using call_id for maintainers' clarity. Comments will be kept up-to-date if code changes.

- **Modular, Reusable Components:** Avoid monolithic functions. Where possible, break down functionality into small, reusable pieces. For example, the file tree item rendering can be a separate component `FileNodeItem` which can be tested independently. The diff size check could be a helper function `isDiffTooLarge(diff: string): boolean` to encapsulate that logic. This not only aids testing but also allows reuse (e.g. if we later add a feature to compare two arbitrary files, we can reuse the diff size guard). New components should be as stateless as makes sense (receive props and callbacks) and rely on external state via props or Zustand rather than hidden internal singletons.

- **Accessibility Considerations:** As we add UI components, ensure they meet accessibility guidelines:

- Use proper ARIA roles and labels. For example, the timeline component likely is a list of events – ensure it's marked up as a list (`<ul>` or appropriate semantic container) and that interactive elements (expansion toggles, filter inputs) have `aria-label` or visible labels. The side-by-side message view should use roles like `article` or `section` for messages or at least ensure screen readers can read the user vs assistant distinction (perhaps visually hidden labels "User message"/"Assistant message").
- **Keyboard Navigation:** All interactive elements must be reachable and operable via keyboard. We will test that one can tab through the filter tags input (able to add/remove tags via keyboard), open the session list modal with a keyboard, navigate the timeline (possibly arrow keys to move between events – we might add this feature if not present), and open/close the file history modal with keyboard (Esc to close, etc.). If the new timeline component from Aceternity UI has internal keyboard support (e.g. arrow key navigation along timeline), we'll integrate that. Otherwise, we may add a custom handler: e.g., when timeline is focused, Up/Down arrows scroll or move selection.
- Color contrast will be respected. If using new color schemes (sparkles or beams might introduce colors), ensure text is still readable. The app already has a contrast checker for theme choices [45]; we will likewise ensure any text on non-standard background (like text over a beam background) meets WCAG AA contrast. If not, provide a semi-opaque background or avoid placing text directly on busy backgrounds.

- We'll also incorporate the **Accessibility polish** items noted in the roadmap as an ongoing goal – e.g., perform an audit using Lighthouse or Axe and fix issues (like missing alt attributes, although here mainly code and text are shown, but for any icon-only buttons, ensure an `aria-label` or `<VisuallyHidden>` text).

- **Performance and Efficiency:** Write code with performance in mind but without premature optimization. Use React best practices: avoid unnecessary re-renders by using `React.memo` for pure components (like individual event cards could be memoized since they only change when their props (event data or filter highlight state) change). Use efficient data structures – e.g. use a Set for quick lookup (changedFiles set) rather than array `.includes` in large loops. The timeline virtualization is critical for perf; if the new timeline component doesn't virtualize, we will implement windowing (e.g. only render a slice of events) ourselves. Ensure diff heavy-lifting (like computing line-by-line diffs or syntax highlighting) is done off the main thread if possible or lazily when a diff is actually viewed.

- Also consider memory: large sessions could be tens of thousands of events – our structures should be mindful (using maps and arrays which are fine, just avoid deeply nested or duplicative storage).

- **Security:** Although this is a local app, we practice good security hygiene. Sanitize any dynamic HTML (e.g. if we ever display content from events that might contain `<script>` tags, we should neutralize them). Monaco by default will neutralize scripts in diff content since it's just text, but if we use `dangerouslySetInnerHTML` anywhere (for maybe rendering Markdown or HTML content in events), we must sanitize it. Ensure that file system access is only used in user-initiated contexts (browser requires this anyway). The app will remain offline-first (no network calls except possibly to a local dev logging endpoint if enabled) [46].

- **Testing and QA Built-in:** Adhere to testing standards (detailed below) as part of code quality. Write unit tests for new utils and possibly snapshot tests for UI. Also, do manual testing with a variety of session files (small, large, with many file changes, with no changes, etc.) to cover edge cases.

- **Git & Documentation:** All code changes will be accompanied by clear Git commits (following Conventional Commits style if used, e.g. `feat(ui): add new timeline component`). We'll update README or docs as needed. For example, if usage or supported features have changed (like new filter capabilities or new optional UI modes), reflect that in the README or a docs/CHANGELOG. This project might plan to be an npm package; in anticipation, we will keep the public API documented (like methods on the Web Component). Internally, maintain a `docs` folder or inline comments for any complex logic. Code should be self-explanatory where possible, but we won't shy away from comments on tricky sections (like patch parsing regex or intricate state flows).

- **Reusable Design Tokens/Styles:** If introducing new styles (colors, spacing) for the UI effects or Park UI components, integrate them into the existing Tailwind theme or CSS variables. For instance, if Park UI's default theme differs, we can customize it to match our app's three color themes (teal, rose, indigo themes as in app). We might extract Park UI's CSS and merge or override it via Tailwind config (if using Panda might be tricky, but we'll find a way to blend). This ensures visual consistency and that if we switch theme modes the new components also adapt (e.g. tooltips and tags might need dark mode styles). All new components should support both light and dark modes – test them in each.

By following these standards, we aim for maintainable, readable code that can be confidently extended in the future (e.g. to add the deferred features or to open-source the project as a library).

Library & Dependency Usage

Several new dependencies and libraries will be introduced or expanded to accomplish the roadmap goals:

- **Shadcn UI / Aceternity UI:** We will utilize Shadcn's component architecture, possibly through the Aceternity UI extension. Aceternity UI provides pre-built magical components (sparkles, beams, etc.) on top of Shadcn [47]. We will install the Aceternity UI package or use their CLI as documented. Likely steps include running `npx shadcn-ui init` (if not already done) and then adding components via `npx shadcn-ui add timeline sparkles ...` configured to use Aceternity's registry. The ROADMAP suggests pre-configuring a registry for expected components [48], meaning we might

point the shadcn CLI to `ui.aceternity.com` to pull those specific implementations. If that is not straightforward, we can manually copy component code from Aceternity UI's examples (the site often provides the JSX) and adapt it. Dependencies: Shadcn typically requires **Radix UI** (for underlying accessibility logic) and **Tailwind CSS**. Our project already has Tailwind, and if Radix is needed we'll add `@radix-ui/react-*` packages for components we use (like maybe Radix Popover for tooltips or menus). We should also install **Framer Motion** if any of the Aceternity components require it for animations (quite possible for things like canvas effects).

- **Park UI & Ark UI:** Park UI components (built with Ark UI + Panda CSS) will be used for Tags Input, Tree View, Toast, Tooltip, etc. [49] [50] . We have options to integrate:

- **Option A:** Install Park UI if it's available on npm (it might not be a single package; it could be a set of code to copy). If available, we add it via `npm install park-ui` (and ensure peer deps like `@ark-ui/react` and `@ark-ui/panda` are installed). Ark UI is headless (like Radix), providing logic for focus management etc., and Panda CSS is a utility for styling (which might conflict with Tailwind if both are used).
- **Option B:** Copy needed source from Park UI's open repository (since it's MIT). For example, copy the TagsInput component code and then replace Panda styling with Tailwind classes manually. This might be feasible because the logic is more important than the styling, and we prefer not adding an entire CSS-in-JS pipeline for just a couple of components.
- We will likely go with Option B to minimize new complexity: incorporate the core of Tags Input and Toast. For tags input, Ark UI provides base logic (maybe use Ark UI's useTagsInput hook or similar) and we style with Tailwind. The outcome is a custom `<TagsInput>` component in our codebase. Same for TreeView: we can use the structure from Park UI but plug it into our Tailwind theme (for example, using heroicons or lucide icons for folder/file icons).
- If time permits, we can also try to use Park UI more directly by enabling Panda. That would involve adding Panda's config (a slight overlap with Tailwind). But given our focus, we lean towards extracting what we need.
- **Toast:** We can use Park UI's Toast which likely comes with a context/manager to display messages. Alternatively, use an existing lightweight library or implement a simple context ourselves (since a toast is essentially a div that appears for a short time). Park UI's advantage is design consistency and possibly nice animations. We'll attempt to use it by copying styles if needed.
- **Tooltip:** Could use Park UI's or just use HeadlessUI's popover with our own style. Preline also has a tooltip script that uses data attributes. Possibly simplest: we can use a small library like Tippy.js for tooltips, but since Park UI offers one and we're anyway using Ark UI logic for tags input, we might keep it consistent and use Ark UI's Tooltip primitive.

In summary, we'll integrate Ark UI by `npm install @ark-ui/react` (for hooks and context it provides). We won't fully adopt Panda CSS globally; instead, integrate necessary styles manually or via Tailwind where possible. We'll note in documentation that these components were derived from Park UI.

- **Preline UI:** Preline provides Tailwind plugins and components, especially useful for the Data Table and possibly Stepper and Accordion [18] [51] . We will add Preline by installing it (`npm install @preline/plugin`) and adding it to the Tailwind config plugins [52] . This will inject necessary utility classes or base styles for components like accordion and tooltips. We also include Preline's JS if needed: the documentation suggests adding a script for certain interactive components (like the datatable sorting might require a small JS snippet). In a React context, we might not need their JS if

we reimplement interactions, but for speed we can include it. The script can be added via a `<script>` tag in index.html or imported in a useEffect (since it likely checks for certain data attributes to enhance elements).
- We'll use Preline's CSS components primarily as examples with our own React wrappers. E.g. create a `<SessionTable>` component that outputs table HTML with classes as per Preline's datatable example, and then the Preline JS (if loaded) will automatically add sorting, etc. If that proves cumbersome, we might implement sorting/filtering ourselves in React (since the data is already in state).

- For Accordion: if we need a simple accordion for grouping events or file history, Preline's HTML snippet + a little JS could work, but since we have React, we might as well implement an accordion component (even via HeadlessUI's Disclosure or Radix Accordion) and just style it similar to Preline's look. So Preline is mainly for design reference and possibly their CSS for consistency with any other Preline elements.

- **Diff Rendering:** We will continue with **Monaco Editor** for diff. It's already a dependency (`monaco-editor` and `@monaco-editor/react`) [14]. We ensure these are up-to-date to benefit from any performance improvements. Lazy loading is set up; we might adjust the strategy to use dynamic `import()` for monaco when first needed (if not already the case). No new diff library is introduced for now, but we will monitor performance – if needed, we might add a smaller diff library for fallback. For example, as an optional dependency we could consider `diff2html` for generating HTML diffs of huge files quickly. This is optional; we will document that large diff handling might be improved by using a lighter renderer if Monaco proves insufficient. However, initial plan is using Monaco with guardrails as described.

- **Other Dependencies:**

- We might incorporate a **highlighting library** for any code blocks outside Monaco (like if we allow copying code or showing preview without Monaco, maybe use Prism.js for syntax highlighting small snippets). But currently, all diff highlighting is via Monaco, so no immediate need.
- For testing (discussed later), ensure **Jest/Vitest** and React Testing Library are set up. If not, we'll add them (dev dependencies).

- If multi-threading needed: possibly use **Comlink** and a Web Worker for heavy computations like the large export or hashing. The roadmap defers large export to a worker [53] – we can lay groundwork by setting up a worker script and using `vite-plugin-worker` or similar to bundle it. Not immediately a dependency to add, but something to consider.

- **Installation & Configuration:**

- After adding new dependencies, update documentation for developers: e.g., "Run `npm install` to get new packages: @ark-ui/react, @monaco-editor/react, etc. If using Preline, ensure to add it to tailwind.config.js plugins." Also, if Shadcn CLI is used to generate components, those components might have their own small dependencies (like lucide-react for icons, or class-variance-authority for styling utilities) – we need to install those as needed.
- Tailwind config needs enabling Preline:

```
// tailwind.config.js
plugins: [
  require('@preline/plugin'),
  // ... other plugins
]
```

And ensure content includes node_modules preline if necessary (some Tailwind setups require content: ["…/node_modules/@preline/*/.js"] etc., but the plugin might handle it).

- We also ensure to enable any theming required by libraries. If Ark UI needs a context provider (some libraries do), we will wrap our app with it. For instance, Ark UI might require `<ArkUIProvider><App/></ArkUIProvider>` if it has global context (to manage portals or z-index). We will check documentation.
- **Potential Conflicts:** Using both Radix (via Shadcn) and Ark UI in one project is unusual since they solve similar problems. We'll be careful to avoid conflicts: for example, if both provide global CSS resets or focus outlines, we might see double. But since Tailwind/HeadlessUI is already used, it might be fine. We should also avoid duplicate functionality – e.g., we won't use two different tooltip libs simultaneously; pick one and stick to it globally for consistency.

- **Bundle Size Consideration:** Adding these UI libs will increase bundle size. We mitigate by cherry-picking components and tree-shaking. Shadcn's approach of copying code means only what we use is included. Ark UI is fairly small (headless logic). Preline is mostly CSS. We will monitor the built bundle: since one distribution mode is embedding on other sites, keeping the size reasonable is important. We might need to trim unused imports (e.g., don't import all of Preline's huge CSS, just the parts we need if possible). Tailwind's purging will drop unused classes anyway.

- **Diff for Large Files Alternative:** For performance, if needed as an enhancement, consider **Google's Diff Match Patch** or a streaming diff algorithm in a Web Worker. We won't integrate it now but keep it in mind. Another tool: **Monaco's DiffNavigator** for better UX on large diffs (already have some Monaco features by default).

In summary, the plan is to **introduce Aceternity UI and Park UI components thoughtfully**, copying/adapting where integration is complex, and to **configure Preline** to take advantage of ready-made Tailwind components. We'll document all new setup steps for future devs. The end result should be a richer UI without sacrificing the project's simplicity (no heavy backend or dramatically new frameworks – everything stays in the React/TS/Tailwind ecosystem).

Testing Strategy

A comprehensive testing approach will be adopted to ensure new enhancements work correctly and reliably, without regressing existing features:

- **Unit Testing:** We will write unit tests for all new pure functions and utils:
- The session parsing (if not already tested) – feed it sample .jsonl lines (including edge cases like malformed JSON, or newer event types) and assert the output `ParsedSession` matches expected structure or that errors are handled gracefully.

- The `analyzeFileChanges` mapping function – create a mock list of events (like the examples given in ROADMAP [54] [32]) and assert that the output maps contain the correct file paths and call_ids. For instance, given an input events array with one FunctionCall (applypatch) and its output, ensure the function returns that file in fileToChanges and the call_id in callToFiles. Include cases with multiple file changes in one call and multiple calls affecting one file to cover merging logic.
- Filter logic – test `filterEvents` with various filter combos. We can simulate a small session: e.g., events of each type, some with certain roles, etc. Then apply a filter (like type=Message, role=assistant) and expect only the matching ones remain. Test the filePaths filter by marking an event as affecting a certain file and see if filter picks it up. This ensures the integration of callId->file mapping with filters works (for that we might call analyzeFileChanges inside the test or mock its result).
- Utility functions for diff handling (if we add any, e.g. a function `isBinary(content)` or `isLargeDiff`) should be tested with sample inputs (like a short text vs some binary data blob).

We will use **Vitest** or **Jest** (depending on project setup; Vitest is likely since Vite project) and keep tests in a `__tests__` directory or alongside modules as appropriate. Aim for a high coverage (target $\geq$ 80% lines for new code).

- **Component Testing:** Use **React Testing Library** to test new React components in isolation:
- TimelineView: we can provide a small list of events as props and verify it renders them in order. If grouping is enabled, test that events are grouped (perhaps by checking that a FunctionCall and its output appear under one container). Since the actual Timeline UI might involve scroll or sticky elements, we may simplify by testing the data transformation rather than pixel values. For interactive parts like clicking an event (if it calls a callback), simulate a click and assert the handler was called with correct event id.
- TagsInput: simulate typing and adding tags. This likely involves fireEvent.change or keyDown on the input and then checking that the onChange callback receives the new tags array. Also test deleting a tag (maybe by pressing backspace or clicking an “x” if rendered).
- FileTree with changed file markers: feed it a file list and a set of changed file paths, then assert that those files have an extra icon or class (we can query by test-id or role for those icons). Also simulate clicking a file and ensure the handler fires with correct path.
- FileHistoryModal: give it a prop with multiple FileChangeEvent items and ensure it renders one section per change (we expect to see e.g. the file name or timestamp in the UI for each diff). If using a carousel, it might render one at a time, so test that clicking “Next” changes the content (could be tricky to test without a DOM environment that supports the animation; possibly we implement carousel as stateful show index, which we can manipulate in test).
- Toast: trigger a toast (maybe by calling a context method or rendering the component with a certain prop) and verify it appears and disappears after timeout. We might need to fake timers for that.

We will also test that these components integrate with Zustand state correctly. Possibly using a custom render that wraps in provider.

- **Integration/UI Testing:** Because this is a user-facing app, we will set up end-to-end style tests, possibly using **Cypress** or **Playwright**. Key user flows to test:
- **Open a session file:** Simulate dropping a known session JSONL or using file input (this is tricky in automated tests, but we can programmatically feed the data by stubbing file picker). Verify that after loading, the timeline appears with expected number of events, the metadata panel shows correct

session ID, and no errors occur. If the session file has known content (we can craft a sample with, say, one function call that adds a file), we verify that the file tree shows that file, etc.

- **Apply filters:** With a session loaded, interact with the tags input – e.g. type "Message" and select the Message type filter tag, ensure the timeline now only shows messages. Add another tag for role "assistant", ensure now only assistant messages visible. Remove tags, verify timeline back to full. Also test the bookmarks filter: bookmark an event (we might simulate clicking the star icon) then click "Bookmarks only" toggle and verify only that event remains.
- **Open diff viewer:** Click on a file's "Open diff" button (or if we have a file that was changed, click it in the tree). The Monaco diff should open in a modal or panel. We check that the content of the diff matches expected (if using a known patch in the test session). Also test toggling split/inline or theme toggle if present – ensure no error and UI updates.
- **File history navigation:** For a file changed multiple times, test that the "history" view works. This could involve clicking a special icon next to the file name. Then verify that a modal or accordion appears listing multiple changes. Possibly simulate clicking the second change and ensure the diff content updates to that version.
- **Exporting data:** Use the UI buttons for Export (JSON, Markdown, etc.). This might trigger a download; in a test environment, we intercept the download event. Alternatively, we call the export function directly and verify the output string contains expected content (e.g. Markdown contains the assistant message text).
- **Session list and workspace:** If feasible, test the All Sessions view: add some sessions to IndexedDB (we might mock IDB or use a memory alternative in tests) and then simulate clicking "View all". Ensure the datatable shows those sessions, test sorting by date if implemented, and loading a session from that view.
- **Visual effects toggle:** If we add a setting to disable effects, test that enabling/disabling it removes or adds certain classes/DOM elements (like sparkles container appears or not).

- **Responsiveness:** Possibly test in different viewport sizes (if using Cypress, can simulate mobile width) to ensure layout (especially the timeline and file tree which might be side-by-side or stacked) still works, and that no component overflows wrongly.

- **Performance Testing:** While not typical in automated tests, we will do some runtime performance checks:

- Load a large session (~5k events, and some big diffs) in a dev environment and use performance profiling (DevTools) to ensure timeline scroll stays above ~45 fps [15] . We can also instrument critical loops (like filtering or mapping) with console.time to see they run in tens of milliseconds, not seconds. If any performance issue is found, address by optimizing or splitting work (e.g. using requestIdleCallback for heavy non-urgent computations like pre-computing diff hash).

- Memory usage for large sessions: ensure no memory leaks. Using React's dev tools to check components unmount properly (like if switching session or closing diff, the components are garbage collected). Our error boundaries or global state should not accidentally retain references (we should clear the session state when unloading if needed).

- **Test Tooling:** The project likely uses a testing framework already (if not, we'll set up Vitest + jsdom for component testing, and possibly Cypress for e2e). We will include these in the devDependencies. We will also set up npm scripts like `npm run test` and `npm run test:e2e`. If CI is in place, ensure tests run on push. We aim for both **automated** tests and thorough **manual QA** for things

that are hard to simulate (like actual drag-and-drop of a file, checking canvas animations – those we manually verify in the browser).

- **Regression Testing:** We will re-run existing tests (if any) to ensure our changes didn't break earlier functionality. For example, if there were tests for the bookmarks or export, they should still pass. If we change any behavior (like how filters are applied with tags input vs old UI), update tests accordingly to reflect the new way (ensuring the underlying outcome is same or improved).

- **Accessibility Testing:** As part of QA, use tools like Axe or Lighthouse to scan the app for accessibility issues after changes. This can be integrated in tests (there are Jest and Cypress Axe integrations that can catch issues). We'll fix any critical ones (like missing labels or low contrast) as part of our DoD (Definition of Done) for each UI change.

- **Example Test Cases:**
  Unit test example for analyzeFileChanges:

```
const events = [
  { type: 'ResponseItem', payload: { type: 'function_call', name: 'shell',
arguments: '...***Begin Patch*** Add File: foo.txt ...', call_id:
'call_ABC'} },
  { type: 'ResponseItem', payload: { type: 'function_call_output',
call_id: 'call_ABC', output: '{"output":"Success. Updated the following
files:\nA foo.txt\n"}' } }
];
const result = analyzeFileChanges(events);
expect(result.fileToChanges['foo.txt']).toHaveLength(1);
expect(result.fileToChanges['foo.txt'][0].action).toBe('added');
expect(result.callToFiles['call_ABC']).toContain('foo.txt');
```

Component test example for TagsInput:

```
it('allows adding and removing tags', () => {
  const onChange = vi.fn();
  render(<TagsInput values={[]} onChange={onChange} />);
  const input = screen.getByRole('textbox');
  fireEvent.change(input, { target: { value: 'FunctionCall' } });
  fireEvent.keyDown(input, { key: 'Enter' });
  expect(onChange).toHaveBeenCalledWith(['FunctionCall']);
  // Now remove the tag
  const removeBtn = screen.getByLabelText(/remove tag FunctionCall/i);
  fireEvent.click(removeBtn);
  expect(onChange).toHaveBeenCalledWith([]);
});
```

We will create similar tests for other components.

By following this testing strategy, we ensure that each enhancement not only works on its own but also plays well with others, and we guard against future regressions as the project evolves. The goal is a stable, high-quality release of these features, supported by automated tests that will continue to validate the application as new changes come in.

Example Usage

To illustrate how the enhanced Codex Session Viewer will be used by both developers and end-users, here are a few scenarios and code snippets:

- **Embedding the Viewer in a Host Page:** Developers can include the viewer as a web component in any HTML page. This remains straightforward even after our changes:

  ```
  <!-- Include the built script -->
  <script type="module" src="dist/element.js"></script>
  <!-- Use the custom element in HTML -->
  <codex-session-viewer style="width:100%; height:100vh;"></codex-session-
  viewer>
  ```

  This will display the UI. All new features (timeline, filters, etc.) are contained within and auto-initialized. For instance, when the user drags a `.jsonl` file onto it, the viewer will load it and show the content with the new interactive timeline and file diffs.

- **Programmatically Loading a Session:** If using the JavaScript API of the web component:

  ```
  const viewer = document.querySelector('codex-session-viewer');
  // After the element is ready/connected:
  viewer.loadSession('/sessions/2025_09_15_run.jsonl');
  ```

  This would trigger an internal fetch and parse (assuming `loadSession` method accepts a path or File). The developer doesn't need to change anything to benefit from the enhancements – the session will load and the timeline will now use the new UI, etc. The output (UI) would look like: a timeline with dated separators, fancy background beams, filter chips at top, etc. (visually improved but functionally similar to before, so existing users aren't confused).

- **Filtering with Tags (UI Example):** Suppose a user wants to filter events to only show function calls that modified `src/App.tsx`. In the new UI, they would click on the filter bar and start typing. As they type "App.tsx", an autocomplete might suggest the file (we can enhance TagsInput to suggest known file paths from the session). The user hits Enter, which creates a tag "`src/App.tsx`". Next, they type "FunctionCall" (or choose from a dropdown of event types) and add that tag. Now two tags are shown: `[src/App.tsx] [FunctionCall]`. The timeline immediately updates to show only events that match both tags. Internally, our filter function used these to filter down to function call events whose call_id mapping includes *src/App.tsx*. This tag UI is much more fluid than toggling

multiple separate controls. *(In code, this could be simulated by setting* `filters.filePaths = ["src/App.tsx"]` *and* `filters.types = ["FunctionCall"]` *and then re-rendering timeline.)*

- **Viewing File Change History:** A user notices that `src/App.tsx` was changed multiple times. In the file tree, this file might have an icon indicating multiple changes. The user right-clicks or uses a new "History" button next to the file name. This opens a **File History modal**. For example, the modal might show:

- "Version 1 – Added at 10:33:10 GMT by call_5OAlcx... (Function: shell apply_patch)" with a diff snippet.
- "Version 2 – Modified at 10:34:44 GMT by call_30hpCmd..." with diff.
- "Version 3 – Modified at 10:34:55 GMT by call_RpmIAER..." with diff.

The user can scroll through or click each version. If they click "Open full diff" on Version 2, it launches the Monaco diff viewer comparing Version 1 vs Version 2 of that file, for detailed review. Under the hood, this uses our `fileToChanges` map and the stored diffs to reconstruct each version (by sequentially applying patches, or simply showing the patch itself if showing differences from previous version). This example highlights how the call IDs and timestamps are exposed to the user for transparency – they can correlate those with the timeline if needed ("Oh, call_30hpCmd corresponds to the commit where feature X was implemented").

- **Using the Two-File Diff Tool:** On the welcome screen, the user can now select "Compare any two files" (if we improved that UI via CollapsibleCard as indicated by patch to `App.tsx` in the logs [55] [56]). For instance, the user clicks "Two-File Diff" section, chooses a file from their workspace folder and another file (or maybe a file from session vs file from workspace). Then clicks "Compare". The Monaco diff viewer opens in a new panel showing differences between those files. This uses the existing TwoFileDiff component (which we ensured is still working after we transformed the UI container into a CollapsibleCard). The example usage for the developer is minimal (it's a built-in feature), but from a code perspective, we maintained the interface of TwoFileDiff while just wrapping it in a collapsible UI.

- **Bookmarks and Export Workflow (User Journey):** The user bookmarks a few critical events (the UI shows a ☆ on those cards). Then the user toggles "Bookmarks only" to review them, then hits "Export Markdown". The resulting Markdown file (downloaded automatically) contains only those bookmarked events, properly formatted [38]. The file name includes "-bookmarks" and maybe filters info (this was existing behavior we kept). Internally, nothing changed for the user here except the UI polish: e.g., the bookmark button might now have a tooltip "Bookmark this event" and a nicer filled star icon when active.

- **Optional Collaboration Mode (Future):** If we envision a future scenario (not in this release) – e.g. two users could live-follow a session – a developer might use a new API method like `viewer.connectToLiveSession(url)` or `viewer.enableCollaboration(roomId)` to join a live feed. This isn't implemented yet, but our design allows adding it later (since the core data model is separate from UI, feeding new events in real-time would not fundamentally change the UI components).

These examples demonstrate typical uses of the enhanced viewer. The improvements, while extensive (new filters UI, new timeline look, etc.), are designed to be intuitive. A developer embedding the component

doesn't need to do anything special to "enable" them – they come with the new version. For developers working on the project, the example code shows how one might call the new functions or use the new components in isolation (for unit tests or storybook):

```
// Example: Rendering TimelineView in isolation (for Storybook or test)
const demoEvents = importSampleEvents(); // imagine we have a sample session
JSON
const fileMappings = analyzeFileChanges(demoEvents);
<TimelineView
    events={filterEvents(demoEvents, { types: new Set(['FunctionCall']) })}
    groupBy="call"
    onEventSelect={id => console.log("Selected", id)}
/>

// Example: Using the TagsInput component logic
<TagsInput values={['Message','role:assistant']} onChange={vals =>
setFilterTags(vals)} />
// The onChange would parse tags like "role:assistant" and update filter state
accordingly.
```

Such usage snippets help clarify how the components are intended to be used in code. We will include some of these in the repository's docs or as comments in the code for future reference.

Overall, both the developer experience (maintaining/extending the code) and the end-user experience (using the app to analyze sessions) are improved by these changes, while preserving the simplicity of integration and usage that existed before.

Optional Future Enhancements

Looking beyond the scope of the current implementation, several features have been identified as future enhancements. These will not be addressed immediately but are worth outlining for long-term planning:

- **Very Large Session Exports via Web Worker:** For extremely large sessions or exports (e.g. exporting 10k+ events to CSV or Markdown), the work could be offloaded to a background thread. In the future, we plan to implement a Web Worker pipeline for exports [53]. This would prevent the UI from freezing during export. The main thread would send the filtered events data to a worker, the worker would stream back partial results or a completion message with the generated file blob. This enhances performance for heavy users but isn't critical until needed (the current export is synchronous and may suffice for moderate sizes).

- **Accessibility Audit & Full Compliance:** While we integrate some a11y fixes now, a **full audit** is slated as a future task [57]. This involves systematically testing with screen readers (JAWS, NVDA, VoiceOver) and keyboard-only usage to identify any remaining barriers. We expect to refine focus order (maybe add shortcuts to jump between timeline and file list), add ARIA descriptions for complex widgets (like describing the timeline's role or the file history carousel). Ensuring every feature is operable without a mouse and clearly understandable via assistive tech will be the goal.

This might also include providing alternatives to visual effects (like the globe or sparkles) or ensuring they are disabled in high contrast mode.

- **NPM Package & Module Export:** To increase adoption, we plan to package the viewer as an installable library [58]. This means publishing to npm (e.g. as `codex-session-viewer` package) so developers can `npm install` and use it in their own React projects or import the custom element easily. We'll need to adjust the build to output not just a bundled script but also modular exports (ESM and possibly CommonJS) for consumption. This might involve refactoring to allow usage as a standard React component (in addition to the web component). We would also set up a CI pipeline for publishing on tag releases. A static site (for example, a GitHub Pages or Netlify demo) will also be prepared as part of distribution [59].

- **Live Session Following:** In the future, the viewer could attach to a live Codex session and update in real-time as new events come in [60]. This "live-follow mode" would require a WebSocket or similar connection to the Codex CLI or a log stream. We would need to handle streaming input (appending to events list, auto-scrolling perhaps). The architecture is already favorable to this (since the timeline can update dynamically), but we'd have to implement a connection manager and possibly a minimal protocol. This is a non-goal currently [61], but kept in mind so nothing in our current design precludes adding it.

- **Plugin API:** Down the line, we envision a plugin system where developers can extend the viewer with custom functionality [60]. For example, a plugin could analyze code diffs to detect certain patterns or integrate with Git to fetch commit history for a file. A plugin might add a new panel or augment event cards with extra info. To enable this, we might expose hooks or an event bus in the app. This is complex and will be done once the core is very stable. Our current refactoring (with modular components and clear data flow) is a step toward making a plugin API feasible.

- **Multi-User Collaboration:** Although currently out-of-scope [62], a collaborative mode could allow multiple people to view the same session remotely, perhaps with one person driving and others following along (like Google Docs for session replay). This would require a server and syncing mechanism (likely far beyond the local-first approach we have). If implemented, it might use something like WebRTC or a small WebSocket server to broadcast state changes (like "user X scrolled to event Y"). We note this as a future possibility, but it would not alter our current code unless we start abstracting state to be shareable. For now, we simply ensure that our state could be serialized if needed (which is partly done via URL hash deep links already).

- **Advanced Analytics and Insights:** Future versions might include analysis of the session data – e.g., detecting anomalies, summarizing the coding session, or providing metrics (like total tokens used, success rate of patches, etc.). The PRD explicitly excludes ML-based insights currently [61], but as Codex and its usage evolve, such features might be valuable. We'd then integrate with some ML service or run local analysis on the events. Our current design should keep data accessible (we already collect token usage events, function call durations, etc., which could feed into analytics).

- **Codex Integration & Editing:** At present, the viewer is read-only and independent of Codex runtime [63]. In the future, one might integrate it with the Codex CLI so that you could click an event and perhaps revert a change or re-run a command in Codex. Or even allow editing a session (correcting or annotating it). These are currently non-goals [61], but if user feedback asks for it, our code should

be adaptable. For example, editing a session log might be as simple as allowing the JSONL to be re-exported after changes – not too hard if we have in-memory representation. But sending commands back to Codex would need an API on Codex's side.

- **Improved Diff Handling for Binary/Images:** We currently skip binary diffs, but a future enhancement could be to render image diffs (e.g., show before/after image side by side if the session produced images) or at least detect image file changes and provide a thumbnail. Also, a nicer UI for large text diffs (like a search within diff, or a way to load diff in chunks) could be built. Possibly integration with Git for diff view (if the project is a git repo, show commit diffs context).

- **Session Comparison:** Eventually, users might want to compare two different sessions (for example, two runs of the agent on the same task). A future feature could allow loading two sessions and showing differences in their outcomes or a side-by-side timeline. This is speculative, but our modular approach (each session parsed separately) means we could instantiate two viewers or build a combined view.

In conclusion, the current development focuses on delivering a solid v3 with improved UI/UX and maintainability. The above future enhancements, such as packaging [58] and collaboration, ensure that the project has a **long-term roadmap** and that the architecture we put in place now will support those expansions when the time comes. Each of these can be tackled incrementally in the future without major overhauls, as long as we maintain clean separation of concerns and a robust core. The roadmap is intentionally "locked" for this phase [64] to prevent scope creep, but it's clear the vision extends beyond, and we've documented these possibilities so that future contributors have a guide for where to go next.

---

[1] [15] [16] [25] [26] [27] [28] [29] [35] [36] [40] [42] [46] [53] [57] [58] [59] [60] [61] [62] [63] [64] prd-v3.md
file://file-R2nmetpS3TkHopKLJsmvtj

[2] [14] [19] [24] [33] [37] [38] [39] [41] [45] README.md
https://github.com/AcidicSoil/codex-session-viewer/blob/8c1dbada4f730a239131aa763167e73babc23cac/README.md

[3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [17] [18] [20] [21] [22] [23] [30] [31] [32] [34] [43] [44] [48] [51] [54] [55] [56] ROADMAP.md
file://file-8s7H9KRMRC4jMUfPZcQs5D

[47] Aceternity UI
https://ui.aceternity.com/

[49] Releases · cschroeter/park-ui - GitHub
https://github.com/cschroeter/park-ui/releases

[50] cschroeter/park-ui: Beautifully designed components built with Ark ...
https://github.com/cschroeter/park-ui

[52] Install Preline UI as a Tailwind CSS plugin
https://preline.co/docs/frameworks.html