**Plan — break the visualizer into small, maintainable units (step‑by‑step)**

I inspected your single-file visualizer V14\_inliner\_Stationary.html to understand how things are currently wired (UI + DOM refs, a large inline <script> with global state, two p5 sketches, IndexedDB helpers, file parsers, sync/animation loop, etc.). Example locations in your file: the global state & DOM refs block, the p5 sketch code and the CAN-log parser.

Below is a careful, beginner‑friendly, **step‑by‑step** plan to refactor the app into smaller modules. Each step has explicit sub‑tasks and verification checks you can follow. Do **not** start refactoring yet — this is a plan only, as requested.

**Quick overview / goals**

* Keep behaviour the same while moving code into separate modules/files.
* Make unit-testable chunks (parsers, utils, state) and isolatable UI pieces (modal, theme, p5 sketches).
* Reduce global variables and centralize side effects (file I/O, DB, DOM).
* Make it easy to maintain, add features, or write tests.

**Project structure I recommend (example)**

/index.html <-- minimal shell, loads compiled JS or ES modules

/styles.css <-- small overrides (Tailwind still via CDN)

/src

/constants.js

/utils.js

/state.js <-- central app state + simple pub/sub

/dom.js <-- DOM selectors + UI update helpers

/modal.js <-- showModal / hideModal

/theme.js <-- theme toggle logic

/db.js <-- IndexedDB helpers: initDB, saveFileToDB, loadFileFromDB

/fileParsers.js <-- JSON parsing + CAN log processing

/sync.js <-- animation loop, frame mapping, resync logic

/p5

radarSketch.js <-- radar p5 sketch (drawPointCloud etc.)

speedGraphSketch.js <-- speed graph p5 sketch

/drawUtils.js <-- helper pure drawing functions and color defs

main.js <-- bootstrap / wiring module

/tests <-- optional: unit tests for utils & parsers

**Step-by-step refactor plan**

Before you start: create a new git branch (e.g. refactor/modularize) and commit the current file. Do small commits after each step with clear commit messages.

**Step 0 — Prep & safety checks (very important)**

1. Commit your current V14\_inliner\_Stationary.html (so you can revert).
2. Create branch refactor/modularize.
3. Run a quick manual smoke test of the current app and note expected behaviour:
   * Load JSON → visualization appears.
   * Load video → video plays, metadata loads.
   * Load CAN log → speed graph appears.
   * Play/pause/stop, timeline scrub, keyboard arrows, toggles.
4. Note any console warnings or errors to watch during refactor.

**Why:** small incremental commits make debugging trivial.

**Step 1 — Minimal index.html + module entry (no logic moved yet)**

Goal: create a minimal HTML shell that will later import JS modules. Do *not* change behaviour yet.

Tasks:

1. Copy the current file to a new folder (project root). Keep V14\_inliner\_Stationary.html as backup.
2. Create index.html containing only your DOM structure (all the HTML markup stays the same) and move the big inline <script> into src/main.js later. For now, keep the inline script where it is but add a comment header describing the planned module split.
3. Ensure Tailwind and p5 CDN links remain (p5 is used globally by your sketches).

Verification:

* Open the page and check nothing changed visually/behaviourally.

**Step 2 — Extract constants**

Goal: remove magic numbers and share them across modules.

Tasks:

1. Create src/constants.js and move constants from the script (e.g., VIDEO\_FPS, MAX\_TRAJECTORY\_LENGTH, radar bounds RADAR\_X\_MIN/MAX, RADAR\_Y\_MIN/MAX) into named exports.
2. Ensure modules import what they need (import { VIDEO\_FPS } from './constants.js').

Why:

* Makes tuning easier and reduces duplicate values.

**Step 3 — Extract utilities (pure functions)**

Goal: isolate pure logic (no DOM/p5) so you can test it.

Tasks:

1. Make src/utils.js and move:
   * findRadarFrameIndexForTime, findLastCanIndexBefore (binary search logic).
   * timestamp parsing/extraction functions: extractTimestampInfo, parseTimestamp.
   * small helpers: clamp, formatTime, mapRange if used.
2. Add unit tests for these functions (very small—just enough to ensure parity).

Caveats:

* These functions are currently used directly in animation + parsers so make sure imports work.

Reference: your binary-search functions and timestamp helpers are in-file now.

**Step 4 — Centralized app state + pub/sub**

Goal: avoid scattered global variables. Provide a controlled state object and events when it changes.

Tasks:

1. Create src/state.js:
   * Export a simple object or class holding fields: vizData, canData, videoStartDate, jsonFilename, videoFilename, isPlaying, currentFrame, etc.
   * Provide getters/setters and an event emitter (tiny pub/sub) so modules can subscribe to changes (on('vizLoaded', handler)).
2. Replace direct reads/writes to globals with state usage. At first you can make state just hold values and keep existing code referencing it; later change to getters.

Why:

* Makes reasoning about data flow easier and prevents accidental global mutation.

**Step 5 — DOM module (selectors + UI helpers)**

Goal: centralize document.getElementById(...) and event wiring in one place.

Tasks:

1. Create src/dom.js:
   * Export constants for DOM elements (the long list you have now: canvasContainer, videoPlayer, loadJsonBtn, etc.). Example list in file: canvasContainer, video-player, timeline-slider, toggles, modal elements.
   * Export helper functions: setFrameCounter(n, total), showFeatureToggles(), setPlayState(isPlaying), updateEgoSpeedDisplay(value) etc.
   * Expose bindUI() function that accepts callbacks (e.g., onLoadJson, onPlayPause, ...) and wires the DOM events to those callbacks.
2. Move direct event listener attachment out of the big script and into dom.bindUI(...) during wiring.

Verification:

* After wiring, check that clicking Load JSON/Video/Can still invokes handlers.

**Step 6 — Modal & theme modules**

Goal: separate the small UI utilities.

Tasks:

1. src/modal.js: move showModal and hideModal and the modal DOM references into this module. Export showModal(message, isConfirm) returning a Promise like current code.
2. src/theme.js: encapsulate theme toggle logic (setTheme, theme button listener), and call subscribed redraws on theme change. Current theme code in file should be moved.

Verification:

* Theme toggle still works and persists to localStorage.

**Step 7 — IndexedDB service**

Goal: single file to manage persistent caching.

Tasks:

1. Create src/db.js. Move initDB, saveFileToDB, and loadFileFromDB into it. Export initDB(), saveFile(key, value), loadFile(key).
2. Replace inlined calls with imports.

Why:

* Isolates browser persistence details and makes mocking easier.

Reference: your current IndexedDB code is in the inline script.

**Step 8 — File loaders & parsers**

Goal: move JSON and CAN parsing + sanitization into a dedicated module.

Tasks:

1. Create src/fileParsers.js:
   * sanitizeJsonString(jsonString) (replace Infinity/NaN with null as you already do).
   * parseVizJson(jsonString) — parse JSON, compute timestampMs for frames (currently in initializeVisualization). Ensure it returns a clean vizData object.
   * processCanLog(logText, videoStartDate) — move CAN regex/parsing logic here and return canData[]. Note your file currently uses a regex and hard-coded CAN ID '30F'. Make canId a parameter or clearly documented constant.
2. Add small tests: give a tiny JSON sample and a tiny CAN log sample and confirm outputs match old behavior.

Verification:

* Replacing calls should not change parsing results.

**Step 9 — p5 sketches into modules**

Goal: separate the two sketches into independent modules that accept state and callbacks.

Tasks:

1. Create src/p5/radarSketch.js:
   * Export a factory: createRadarSketch(containerElement, getStateCallbacks, options).
   * Move your big sketch function there (everything inside let sketch = function(p) { ... }). Make it rely on passed getState() (a function that returns the latest vizData, currentFrame, toggles), rather than global vars. The sketch should return an object with .redraw(), .resize(), and .destroy() (to allow re-instantiation).
   * Break large draw helpers into separate functions in drawUtils.js and import them.
   * Keep p5-specific buffers inside the module (e.g., staticBackgroundBuffer).
2. Create src/p5/speedGraphSketch.js similarly for the speed graph (speedGraphSketch), with methods .setData(canData, vizData, duration), .redraw().
3. Important: p5 is a global loaded via CDN, so these modules call new p5(...) when created (same as now).

Caveats & notes:

* Ensure you do **not** create multiple p5 canvases on the same DOM container without calling .remove() on the old one.
* Provide the sketch modules a themeChanged() or have them listen to the state pub/sub to redraw on theme toggle.

Verification:

* After moving, radar and speed graph behave exactly the same (visuals unchanged).
* Confirm toggleSnrColor, toggleStationaryColor behavior works (you have a new toggle for stationary color in your file).

**Step 10 — Drawing helpers & color definitions**

Goal: make the drawing functions re-usable and isolated.

Tasks:

1. src/drawUtils.js: move out functions like drawPointCloud, drawTrajectories, drawTrackMarkers, and drawSnrLegendToBuffer as pure-ish functions accepting:
   * p5 instance p,
   * data (points / tracks),
   * options (color mode, min/max SNR, toggles),
   * plotScale values.
2. Export color constants (stationaryColor, movingColor, clusterColors set) so tests or other modules can reference the palette.

Why:

* Easier to tweak visuals without touching p5 scaffolding. Also improves testability of logic (when possible).

Reference: your current draw functions live inside the sketch draw() code.

**Step 11 — Synchronization & playback control**

Goal: centralize animationLoop, updateFrame, resync logic.

Tasks:

1. Create src/sync.js:
   * Export startPlayback(), pausePlayback(), stopPlayback(), seekToFrame(frameIndex, {forceVideoSeek}).
   * Move animationLoop, updateFrame, resetVisualization, updateCanDisplay, updateDebugOverlay here.
   * Use state and DOM helpers instead of globals.
   * Keep the same resync heuristics (drift > 0.15s resync), and the offset handling (use offsetInput via dom helpers).
2. Ensure this module publishes events (frameUpdated) on the state pub/sub.

Why:

* Keeps time calculations inside one place and avoids duplicated logic.

Reference: animation/sync code currently is inline.

**Step 12 — Main wiring / bootstrap**

Goal: small main.js that ties modules together.

Tasks:

1. src/main.js does:
   * import db.initDB, dom.bindUI, fileParsers, state, createRadarSketch, createSpeedGraphSketch, and sync.
   * initialize DB, load cached files (like current DOMContentLoaded logic), and call finalizeSetup() once video metadata ready — same flow you have now but orchestrated through modules. There is code handling loadFileFromDB and Promise.all currently — move that logic here.
   * call dom.bindUI({onLoadJson: ..., onLoadVideo: ..., onPlayPause: ...}) so DOM module doesn't need to know internals.
2. Keep this file small: it should only wire modules and pass dependencies.

Verification:

* App behavior unchanged. All UI controls function.

**Step 13 — Testing checklist after each major step**

After each step above, run this small checklist:

1. Does the page still load with no JS errors?
2. Can you load a sample JSON and see radar points?
3. Can you load the video and see it play and sync when you Play?
4. CAN log: load a sample CAN log and see the speed graph / CAN display.
5. Toggle features: SNR color, cluster color, inlier color, stationary toggle (new toggle) — visuals update.
6. Keyboard arrows for frame stepping still work.
7. Theme toggle still works and causes redraws.

Make a commit labelled with the step name on success.

**Step 14 — Performance & robustness improvements (after modularization)**

Once code is modular:

1. Throttle/debounce slider & resize handlers.
2. Avoid unnecessary full redraws — only redraw p5 when needed (you already use .noLoop()/.redraw() in places; keep that pattern).
3. Add defensive checks for large JSON files: show a progress indicator or chunk parsing if necessary.
4. Consider lazy loading heavy modules (p5 sketches) only when vizData is loaded.

**Step 15 — Documentation & developer friendliness**

1. Add README.md describing:
   * project structure,
   * how to run locally (a simple static server e.g., python -m http.server or npx serve),
   * where to change radar bounds or CAN ID (document constants).
2. Add inline JSDoc to exported functions so future devs know inputs/outputs.

**Step 16 — Optional enhancements (nice to have)**

* Move to a bundler (Vite/Parcel) for nicer dev workflow and to allow ES module imports of p5 (optional). If you keep CDN p5, your modules can still call new p5(...).
* Add TypeScript or JSDoc types for critical modules (utils, parsers).
* Add unit tests for parsers and binary-search functions.

**Things to watch for (pitfalls & gotchas)**

* **p5 instances**: creating multiple p5 canvases without removing old ones will leak canvases — ensure .remove() or keep a single instance per container.
* **Video metadata timing**: you rely on video.onloadedmetadata in the original code; keep that behavior so video duration is known before building speed graph.
* **CAN parser regex** is currently hard-coded to ID 30F — make that explicit in constants or fileParsers.
* **Global toggles**: the UI adds new toggles (e.g. toggle-stationary-color) — ensure that the toggles array / mutual-exclusion logic moves with the DOM module.

**Short acceptance checklist (what we will check after full refactor)**

* Everything from Step 13’s checklist passes.
* git diff shows many small modules instead of one big inline script.
* No functional regressions.

**Confidence & assumptions**

* **Confidence:** high for structure & steps — I mapped each major responsibility from your existing file to a module in the plan. I inspected the actual code for global state, p5 sketches, DB and parsing to guide the split.
* **Assumptions:** you want to keep the UI markup the same and keep using CDN p5 for now; you’re ok running a simple local static server for ES modules (if you choose ES modules). If you prefer a bundler, I can include a Vite setup in the next step.