**Persistent Pattern Detector – Product Requirements Document (PRD)**

**Purpose**

The **Persistent Pattern Detector** is a client-side web application designed to identify and visualize repeating patterns within a given dataset or input, entirely in the browser. The app’s primary purpose is to help users quickly detect recurring sequences (e.g., repeated substrings in text or sequences of numbers) without any server-side processing. This PRD focuses on development specifications for the application, detailing exactly what needs to be built using only HTML, CSS, and JavaScript, and ensuring the output can directly inform high-fidelity web app development.

**Scope and Assumptions**

* **Client-Side Only:** The application will be implemented purely on the client side with no backend or database. All processing and state management occur in the user’s browser. There are **no server calls or external data storage**; persistence is achieved via browser storage if needed.
* **Data Types:** The app will primarily handle text input (which can represent sequences of characters, words, or numbers). The pattern detection logic will treat the input as a sequence of tokens (characters by default). No binary files or images are processed (out of scope).
* **Target Users/Usage:** This tool is intended for end users such as students, developers, or analysts who want to find repeated patterns in their data. (No marketing personas are defined here; this is purely a functional tool.)
* **Platforms:** The application must run on modern web browsers (Chrome, Firefox, Safari, Edge) on both desktop and mobile devices. It will be built as a responsive single-page application that works without installation (optionally, it could be packaged as a static site or even a Progressive Web App in the future, but server-side installation is not required).
* **Out of Scope:** Features like user accounts, cloud sync, or any integration with a backend service are not included. Also, no GTM (go-to-market) or marketing features are considered in this document.

**Core Features**

* **Input of Data for Pattern Detection:** Users can enter or upload data (e.g., paste text or numbers) into the app. This will typically be done via a multiline text area or a file upload for a .txt file. The input can be any length up to practical limits (for example, a few thousand characters).
* **Pattern Detection Algorithm:** The app will analyze the provided input to find **repeating patterns**. A pattern is defined as a sequence of characters (or tokens) that occurs more than once in the input. By default, the app will look for repeated substrings of a minimum length (e.g., 2 characters or more, to avoid trivial single-character matches). The detection should handle overlapping occurrences and multiple distinct patterns.
* **Results Display with Highlights:** After analysis, the app will display the results to the user. This includes:
  + **List of Detected Patterns:** A list (or other visual representation) of patterns that were found at least twice. For each pattern, the app may show additional info like the number of occurrences and positions.
  + **Highlighting in Input Text:** The occurrences of each pattern in the input data will be highlighted (e.g., colored background or underline) to visually indicate where the pattern appears. The user can easily see how the pattern is distributed in the data.
* **User Interaction with Results:** The interface will allow users to interact with the detected patterns:
  + Clicking on a pattern in the list will scroll or focus the view to the occurrences in the text (or highlight them more prominently), helping the user locate that pattern.
  + If multiple patterns are detected, the user can toggle which pattern’s highlights are active (especially if patterns overlap, perhaps one at a time or using distinct highlight colors).
* **Controls for Analysis:** The app provides controls such as:
  + A **“Detect Pattern”** button to initiate analysis of the current input.
  + A **“Clear” or “New Analysis”** button to reset the input and results to start over.
  + Optionally, settings controls like **minimum pattern length** or a **case sensitivity toggle** (e.g., treat “ABC” and “abc” as the same pattern or not) to refine what is detected.
* **Persistence of Data (Local Storage):** The app will utilize browser local storage to persist information between sessions. For example:
  + The last entered input data can be saved so that if the user revisits or refreshes the page, their data (and possibly results) remain available.
  + User preferences (like the above-mentioned minimum length or case sensitivity setting, if exposed) will be stored so that the app remembers them next time.
* **Responsive and Accessible UI:** The design will be responsive to different screen sizes and follow basic accessibility practices:
  + On desktop, the input and results can be shown side by side (if space allows); on mobile, they may stack vertically.
  + The app will ensure keyboard accessibility (e.g., the user can tab to the “Detect” button and press Enter, etc.) and readable contrast for text and highlights.
* **No External Dependencies Requirement:** All functionality must be implemented with HTML, CSS, and JavaScript. (It’s acceptable to use lightweight JS libraries for convenience if needed, but no frameworks that require a build step or server.) The app should **function entirely offline** after the initial load, since all logic is client-side.

**User Flows**

This section describes how a user will interact with the Persistent Pattern Detector step by step in common scenarios:

**1. Detecting Patterns in New Input**

1. **Launch Application:** The user opens the Persistent Pattern Detector in a web browser. The main interface loads, showing an empty input area and a “Detect Pattern” button (the results section might be empty or hidden initially).
2. **Enter Data:** The user inputs the data in which they want to find patterns. This could be done by typing or pasting text into the text area. (Alternatively, if a file upload feature is provided, the user selects a text file, and the file’s content is displayed in the text area.)
3. **Initiate Detection:** The user clicks the **“Detect Pattern”** button (or presses an equivalent keyboard shortcut or the Enter key if the UI provides that convenience).
4. **Processing:** Immediately after clicking, the app might provide feedback that it is working:
   * The “Detect Pattern” button may show a loading indicator or become disabled to prevent multiple clicks.
   * A spinner or progress message may appear if the dataset is large, indicating analysis is in progress.
5. **View Results:** Once processing is complete, the results section populates:
   * The list of detected patterns (if any) is displayed, each pattern possibly in a separate row or card, showing the pattern text and the count of occurrences.
   * The input text is now augmented with highlights on each occurrence of each pattern. For example, all instances of the first pattern might be highlighted in yellow. If multiple patterns are listed, each could have a distinct color or there could be an indicator to highlight one pattern at a time.
   * If no repeating pattern is found (i.e., all sequences in the input are unique or below the minimum length threshold), the app will display a message like “No repeating patterns found” in the results area.
6. **Interpret Results:** The user can scroll through the input area to see highlights. They may click on a specific pattern from the list to focus on it:
   * The UI might then scroll the input to the first occurrence of that pattern or momentarily accentuate all its occurrences (e.g., flash or bold the highlight) to draw attention.
   * If pattern highlights are shown one-at-a-time, selecting a pattern might hide highlights for others, making it easier to focus.
7. **Adjust and Repeat (Optional):** The user may decide to adjust settings or input and run again:
   * For example, if there’s an option to ignore case or change minimum pattern length, the user toggles those and then re-clicks “Detect Pattern” to re-run the analysis on the same input with new criteria.
   * The results will update accordingly.
   * Or, the user edits the input (maybe adds more data or corrects something) and then clicks “Detect Pattern” again, yielding new results.

**2. Starting Over / New Analysis**

1. **Clearing Data:** After completing one analysis, the user might want to analyze a completely different dataset. The user clicks on the **“Clear”** (or “New Analysis”) button.
2. **Reset State:** The app clears the text area (or resets it to empty) and removes any results from the previous analysis:
   * All highlights are removed from the text display.
   * The patterns list is cleared (or the results section is hidden).
   * Any status messages or “no patterns found” messages are cleared as well.
3. **Fresh Input:** The user can now input new data (as in the first flow) and click “Detect Pattern” to perform a new analysis. The flow then proceeds as described above for viewing results.
4. **Alternate Approach:** In some designs, starting a new analysis might be as simple as deleting or replacing the text in the input area and clicking detect again (without an explicit “Clear” button). In either case, the user is able to begin a new pattern search easily.

**3. Returning to the App (Persistent State)**

1. **Automatic Restoration:** If the user has previously used the app and then returns (reopens it in the same browser), the application checks for any saved state in local storage. If found:
   * The input area is pre-populated with the last dataset the user entered.
   * The app may automatically re-run the pattern detection on that data on load **or** simply wait for the user to click “Detect Pattern” again. (This behavior will be defined for the best user experience; for example, automatically showing last results might be convenient.)
   * Any user preference (like min pattern length, etc.) is restored to the last used values.
2. **User Awareness:** If data was restored, the UI should indicate this (e.g., showing the highlights and patterns from the last session or a message “Restored last analysis”). The user can then interact with it as before.
3. **Continuing or Clearing:** The returning user can choose to continue exploring that data or clear it to start fresh. All functionalities from the prior flows remain available (they can edit the input and re-run detection, etc., even on restored data).

These flows cover the typical usage. In all flows, the transitions are seamless with immediate visual feedback to keep the user informed.

**User Interface (UI) Components and Screens**

The Persistent Pattern Detector is essentially a single-page application, consisting of a main view where all interaction happens. For clarity, we describe the UI in terms of logical sections/components:

**Main Screen – Pattern Detection Interface**

This is the only primary screen of the app, containing the input area, control buttons, and results display. Key UI components on this screen include:

* **Header Area (Optional):** A simple header bar with the application name “Persistent Pattern Detector”. This gives context to the user. It might also contain a brief description or tagline (“Find repeating sequences in your text”) and possibly an info icon or help link if needed (e.g., to open instructions on usage).
* **Input Section:** This section allows the user to provide data for analysis.
  + *Text Input Area:* A large, scrollable text area (HTML <textarea>) where the user enters or pastes their text/sequence. It should support a substantial amount of text (multiple lines). A placeholder text may guide the user (e.g., “Enter text or sequence here…”).
  + *File Upload (Alternative input method):* Optionally, an upload input (HTML <input type="file">) to select a text file. If a file is loaded, its contents populate the text area. This component is not mandatory but can be included for convenience.
  + *Parameters/Settings:* If the app exposes settings like “Minimum pattern length” or “Case sensitive vs. insensitive”, these controls would be placed near the input. For example:
    - A numeric input or slider for minimum pattern length (e.g., set to 2 by default, allowing range 1–10).
    - A checkbox or toggle for “Ignore case” (when checked, “A” and “a” are treated the same in pattern matching).
    - These settings controls should be labeled clearly and have tooltips or help text explaining their effect.
* **Action Buttons:** The controls to initiate or control the detection process.
  + *Detect Pattern Button:* A prominent button labeled “Detect Pattern” (or just “Detect”) that triggers the analysis. It should be placed near the input area (either below it or in a toolbar). The button should be styled to stand out (e.g., a bright color) and be a logical next step after entering input.
  + *Clear Button:* A smaller button labeled “Clear” or “New Analysis” to reset the input and results. This could be next to the Detect button. This button is disabled (or hidden) when there’s no content to clear (i.e., initially).
* **Results Section:** This section is where results are shown after detection.
  + *Patterns List:* A list or panel that enumerates each detected pattern. This could be a simple vertical list of items, a table, or even cards. Each item at minimum shows:
    - The pattern (exact substring) itself, possibly highlighted or in quotes.
    - The number of times it occurs in the input (occurrence count).
    - (Optional) The length of the pattern, if relevant (or other metrics like pattern frequency percentage).
    - If many patterns are found, the list might be sorted by some criteria (e.g., by occurrence count descending, or by length descending) to show the most significant patterns first. It could also be segmented (e.g., “Top patterns” vs others).
    - Each pattern item in the list is interactive (clickable). There may also be a visual indicator correlating it to highlights (for example, a colored dot or icon that matches the highlight color used in the text).
  + *Highlighted Input Display:* The original input text is displayed with highlights on all instances of each detected pattern. This can be accomplished in a couple of ways:
    - Easiest approach is to reuse the same text area for input and apply background color styles to the text (which is tricky in a plain textarea). Alternatively, after detection, the input section might switch from an editable textarea to a non-editable, styled text block (like a <div> or <pre>) where the matching substrings are wrapped in <mark> or <span> tags with highlight CSS classes.
    - Each distinct pattern could have a unique highlight color. For accessibility, ensure colors have sufficient contrast with the text, or use additional indicators (like underline styles or tooltips).
    - If overlapping patterns exist, the highlight scheme might choose one pattern’s highlight preferentially or use a pattern (like stripes) to indicate overlap. (This is a complex scenario; the implementation can define a reasonable strategy or simply highlight the first pattern and then subsequent non-overlapping instances for clarity.)
  + *Result Messages:* If applicable, this section can show messages:
    - “No repeating patterns found” message when appropriate.
    - Error or info messages (for example, “Input is too large to process” if we impose a limit, or “Please enter at least X characters” if input too short).
    - These messages should be styled distinctly (maybe italic or different color) and appear in the results area.
* **Footer (Optional):** A small footer could contain a disclaimer (“All analysis is done in your browser. No data is sent to a server.”) to reassure privacy, or links to a project repository/info if relevant. This is not a core functional component, just informational.

**Layout:** On a desktop or wide screen, the Input Section and Results Section might be arranged side by side in columns. For example, the left side containing the text input and the Detect/Clear buttons, and the right side showing the patterns list and maybe below it the highlighted text (or vice versa: input on top, results below). On a mobile or narrow screen, these would stack vertically: input area and detect button come first, then below that the results list and highlighted text (with perhaps some spacing or accordion-style toggle if the results are very tall).

**Styling considerations:** The UI should be clean and minimalist, emphasizing function:

* Use a monospace font for the input and pattern display if dealing with technical sequences (optional, but can help alignment and clarity especially if numeric sequences).
* Highlight colors and overall color scheme should be chosen for clarity (light backgrounds for highlights on dark text, etc.). Possibly include both light and dark mode CSS if feasible, or at least one that is easy on eyes for reading lots of text.

No separate “pages” are needed beyond the main analysis interface. Any auxiliary information (help, about) can be provided via modals or collapsible sections to avoid leaving the main page. For instance, clicking a help icon might overlay instructions, but that’s within the same page context.

**Detailed Interactions**

This section breaks down specific interactive elements and their expected behavior in detail, to guide development of a smooth user experience:

* **Typing into the Input Text Area:** As the user types or pastes content:
  + The app should allow free text entry with no real-time blocking. (If very heavy real-time analysis was considered, we’d debounce it; however, by default we will only analyze on demand when the user clicks the button, to keep typing fluid.)
  + If we want an enhancement: a character count or indicator could be shown to the user (e.g., “Characters: 120” or “Length: 120”) to know how large their input is. This is optional.
  + If the user enters data and has previously run a detection, the old results (highlights and patterns list) might either clear automatically to avoid confusion or there should be an indication that these results are from the previous content. Likely, we will clear results whenever the input is changed after a detection, prompting the user to click “Detect” again for new results. (This can be done by clearing the highlights and hiding the patterns list as soon as the user starts editing again.)
* **“Detect Pattern” Button Click:** When the user clicks the detect button:
  + **Validation:** The app will first validate the input. For example, if the text area is empty or only contains whitespace, the app will not proceed. Instead, it can show an inline error or disabled state:
    - If empty: maybe shake the text box or show a small red text “Please enter some data to analyze” near the button.
    - If input is below the minimum length (e.g., only 1 character and min length is 2): show a message “Not enough length for patterns. Please enter more data.”
  + **Disable Inputs:** If validation passes, as the analysis runs, the input area and the detect button should be temporarily disabled or read-only (to prevent the user from altering data mid-analysis or clicking twice). The button might change to a “Detecting…” state.
  + **Loading Indicator:** If analysis is near-instant for small input, a loading indicator might not be necessary. For larger inputs (say thousands of characters), showing a spinner icon or progress bar is helpful. We could simply have a spinner overlay on the results section or an inline “Analyzing…” message. If feasible, a progress bar could be shown if the algorithm can report progress (not required, a generic spinner is fine).
  + **Computation:** The pattern detection logic runs (in the main thread or a Web Worker as needed – see Performance considerations). During this time, the UI should remain responsive (if using a worker, the user could even scroll the input while waiting).
  + **Displaying Results:** Once complete, the app populates the results:
    - The patterns list is generated in the DOM. Each pattern item is created with the text and count. They should be clickable (e.g., <button> or at least have an onclick handler for accessibility).
    - The input text display is updated to show highlights. If using the same textarea for simplicity, we might not be able to color specific text inside it, so likely we convert the text to a <div> with inner HTML wrapping matches in spans. This content replacement should preserve the text layout and be scrollable. We must ensure that after showing results, the user can still copy text from this area (so it should not be an inaccessible canvas or image, it should be actual text).
    - If using distinct highlight colors for multiple patterns, we might include a legend or simply color-code the pattern text in the list to match the highlight color in the text (e.g., each pattern list item has a colored bullet or background).
    - The “Detect Pattern” button returns to active state (enabled) in case the user wants to run it again (like after adjusting something).
    - If any error occurred during processing (e.g., ran out of memory for extremely large input), an error message is displayed instead of results.
* **Clicking on a Pattern in Results:** When the user selects a pattern from the list:
  + That pattern’s occurrences in the text are emphasized. For instance, the highlights for that pattern might glow or a thicker outline is added, or non-selected pattern highlights could temporarily dim.
  + The app may scroll automatically to the first occurrence of that pattern in the text display. If the text area is very long and the pattern is far down, this auto-scroll improves usability. We should animate the scroll for clarity (so the user can follow along).
  + If the pattern list has an active selection state (e.g., the clicked pattern item gets a highlight to show it’s selected), clicking it again could toggle off the selection (returning to the default view showing all patterns equally).
  + If only one pattern’s highlights are shown at a time (an optional design choice to reduce color clutter), then selecting a pattern would hide highlights for the others. In that case, there should be a clear way to return to “show all” (maybe a “Show all patterns” toggle or clicking an already selected item again re-displays all).
* **Using the Clear Button:** On clicking “Clear”:
  + The input textarea is cleared out (value becomes empty string). This can also reset any file input if used.
  + The results list is removed from the DOM or hidden.
  + Any highlights in the text display are removed. If we had replaced the textarea with a div for highlighting, we might swap back an empty textarea for new input.
  + The detect button may be disabled again until new input is provided (optional UX decision).
  + Any messages (error/info) are cleared.
  + Essentially, the app returns to its initial state.
* **Adjusting Settings Interactions:** If settings like min length or case sensitivity are available:
  + Changing a setting should mark current results as outdated. We might automatically re-run detection on setting change, but it’s safer to require the user to click “Detect” again, especially if computation is heavy. So we can:
    - Enable a subtle indication like an asterisk or text “(modified settings, please re-run detection)” if settings change post-analysis.
    - Or automatically clear results when a setting changes to avoid confusion with old results.
  + The UI controls themselves (slider, checkbox) should be easy to change. For example, if the user drags the min length slider from 2 to 3, it can immediately reflect that in a label. If auto-run is off, no immediate analysis is done until they hit Detect again.
* **Responsive UI Behavior:** Interactive elements need to adapt on different devices:
  + On mobile, tapping the text area should bring up the keyboard and the layout should accommodate (maybe the results section collapses or moves below out of view while the keyboard is open, depending on OS behavior). We should test that the highlights and content are still visible after keyboard appears (some mobile browsers shrink the viewport).
  + Buttons should be sufficiently large and spaced for touch input.
  + The pattern list on a narrow screen might become a swipeable carousel if it’s wide, or just a vertical list that scrolls separately. Make sure scroll behaviors don’t conflict (the input text and the pattern list might each be scrollable regions).
* **Keyboard and Accessibility:**
  + The user can press Tab to focus the “Detect Pattern” button after typing in the text area, then press Enter to activate it. We ensure this works by proper HTML structure (e.g., using a <button> element).
  + Pressing Enter inside the text area might by default just create a newline (which is good). We could optionally allow a shortcut like Ctrl+Enter to trigger detection, and document that in a tooltip or help section for power users.
  + Screen readers should be considered: for example, after detection, the list of patterns should be announced as new content. Use ARIA live regions for result announcements (e.g., a live region that says “3 patterns found”).
  + If highlights are purely visual, we might need an alternative for non-visual users, such as listing out the context of each occurrence. However, given this is a dev-focused tool, we assume visual use primarily, but we strive for basic accessibility where feasible.

In summary, each interactive element’s behavior is carefully defined: buttons provide expected actions with feedback, list items are clickable for additional functionality, and changes in state (input, settings) trigger appropriate UI updates or prompts. All these interactions happen instantaneously on the client-side, providing a smooth single-page experience.

**Animations and Transitions**

While this tool is utility-focused, adding subtle animations can enhance usability and clarity. All animations will be implemented with CSS transitions or lightweight JavaScript, keeping them performant. Key animations/transitions include:

* **Loading Feedback:** If a spinner or loading overlay appears during pattern detection, it should fade in/out smoothly rather than abruptly appear. For example, a CSS fade for 0.3s when showing or hiding the “Analyzing…” overlay.
* **Highlight Appearance:** When displaying the detected pattern highlights in the text:
  + Use a brief highlight flash effect: e.g., newly highlighted text could momentarily flash or pulse (transition from a very bright highlight color to the normal highlight color over 0.5s) to draw the user’s eye to the matches.
  + If using an animated scroll to a pattern occurrence when selected, smooth-scroll (CSS scroll-behavior: smooth or JS animate) rather than jump instantly.
* **Pattern List Entrance:** The list of patterns can animate into view. For instance, each pattern item could slide in from the side or fade in sequentially. This gives a visual cue that these items are newly generated. Keep it subtle and quick (e.g., 0.2s delay between each item’s appearance).
* **Transitions on Layout Change:** If the interface switches from the textarea to a highlighted text display (replacing one DOM element with another), use a transition:
  + Perhaps fade out the raw text area and fade in the highlighted text block, to avoid a jarring swap. This could be achieved by layering them and crossfading.
  + On clearing, do the reverse: fade out the results, then make the input editable again.
* **Button Active States:** Use CSS hover and active states on buttons. For example, the “Detect” button might slightly darken on hover or press to provide tactile feedback. On mobile, ensure there’s a visible tap highlight or ripple (one could implement a material-design ripple effect in pure CSS/JS for a nice touch).
* **Error/Info Messages:** If an error message appears (like “Input too short”), it can animate (slide down into view or fade in) near the relevant field, and possibly shake the input box lightly to draw attention (a common UI affordance for invalid input). This should be used sparingly and only for important validation issues.
* **Responsive Transition:** When resizing the window or rotating a device, if the layout is shifting (e.g., from side-by-side to stacked), use CSS transitions to reposition elements smoothly rather than just snapping. This could involve media-query-driven transitions where possible.

All animations should be lightweight and **non-intrusive** – they should enhance understanding (e.g., guiding focus to new content) and not slow down the user. The implementation will use hardware-accelerated CSS properties (like opacity and transform) for smooth performance. Users who prefer reduced motion (if the OS setting is detected) can be respected by disabling non-essential animations via a media query (prefers-reduced-motion) – this ensures accessibility for motion-sensitive users.

**State Management (Client-Side)**

Since this is a client-side application, state management is crucial to handle the dynamic data (input and results) and UI status. The following outlines the state considerations:

* **Application State Variables:** The app will maintain in-memory JavaScript objects/variables for:
  + **User Input Data:** The text that the user has entered. This is essentially mirrored by the content of the input textarea (or replaced content in the highlight view). We may keep a JS variable copy of it for processing (rather than reading directly from the DOM each time) to ease algorithm implementation.
  + **Detected Patterns Result:** After analysis, the list of detected patterns (with their counts and possibly positions). This can be an array of objects in JS, e.g., [ { pattern: "AB", count: 3, positions: [0, 5, 10] }, … ]. This state drives the rendering of the patterns list and the highlights. If the user selects a pattern, we might store which pattern is currently selected (an index or reference).
  + **Settings State:** Any adjustable parameters (like min length, case sensitivity). These are part of the state so that the detection algorithm and UI can use them. For example, { minLength: 2, caseSensitive: false }.
  + **UI State:** This includes things like whether results are currently displayed, whether a loading spinner is visible, or if an error message is shown. It might also include the layout mode (e.g., an indicator if we are showing the editable textarea vs the highlighted text view, though we could manage that simply via DOM).
* **State Transitions:** The application will move through states such as:
  + “Idle” (no data or waiting for user input),
  + “Ready to Analyze” (user has entered data, can press Detect),
  + “Analyzing” (running the algorithm, loading indicator on),
  + “Results Shown” (analysis complete and results displayed),
  + possibly “Error” (if something went wrong). We won’t make an explicit state machine for the user, but internally we ensure that at each state, the UI reflects correctly (for instance, only show loader in Analyzing state, etc.).
* **One-Page State Management:** All the above state is managed within a single page context. We do not need to worry about preserving state across multiple pages or routes since everything happens on one screen. If we use any front-end patterns (like MVC or MVVM), the input data and results can be the Model, and the DOM is the View, updated via JS controller logic after analysis. We can implement this with plain JavaScript by manipulating the DOM directly or by using a library for reactivity (optional, but given simplicity, direct DOM updates are fine).
* **Local Storage Persistence:** When the user performs actions, certain state elements will also be saved to localStorage to persist across sessions:
  + On each successful detection, or whenever input changes, we can save the current input text to localStorage["ppd\_lastInput"] (for example). We might also save the last results or not. Since results can be re-derived by running the algorithm on the input, storing results isn’t strictly necessary – we can recompute on page load if needed.
  + Save user settings (e.g., localStorage["ppd\_minLength"] = 3 if they changed it).
  + On page load, the app will check for these keys. If present, load the values into the state and update the UI accordingly (prefill the text area, set the slider to that length, etc.).
  + Note: localStorage typically allows ~5MB of data per domain. This is plenty for text input persistence (we will also encourage reasonable input sizes). We should handle the case where storing might fail (extremely rare unless user tried a massive input, then we just might not persist that).
* **No Server Communication:** All state stays in the browser. If the user opens the app in a different browser or machine, it will not have their previous state (no syncing). State is also isolated per device/browser (which is expected given no backend).
* **Memory Management:** The app should be mindful of memory since large strings can be heavy. We’ll avoid duplicating the input data unnecessarily. For example, if reading from a file, we insert directly into the text area and maybe keep one copy in JS for analysis. The pattern results will at most hold as many entries as patterns found (which is usually not enormous relative to input size, except pathological cases where the input is highly repetitive).
* **State Consistency:** Ensure that state changes are atomic where needed. For instance, when analysis completes, update both the patterns list and the highlights in one logical operation before rendering to avoid interim states (we don’t want the list to show before highlights are ready or vice versa). This might involve constructing all result DOM off-screen or in memory, then injecting into DOM once complete.
* **Error State Management:** If an error occurs during analysis (say, algorithm runs into an unexpected condition), we should catch it and update state to an “Error” state with an error message, rather than leaving the app in a broken halfway state. The user can then likely continue (maybe clear or modify input and try again).
* **Concurrency:** Since we disable input while analyzing, we don’t expect overlapping analyses. If we did allow it via web worker, we must be careful not to allow multiple workers on the same input at once. Simpler is to block multiple triggers until the first is done.
* **Developer Note:** Given the app’s simplicity, a state management library is not necessary. However, clear organization of state in code is important for maintainability. Comments in code can reference these state variables and transitions as described.

**Data Handling**

This section details how data (particularly the user-provided input and the detected patterns) is handled within the application:

* **Input Data Format:** The primary input is a string of characters. It could be plain text (with spaces, punctuation, etc.), or a sequence of numbers or words. For our pattern detection purposes:
  + We will treat the input as a continuous sequence of characters by default. This means “ABA” is a sequence of characters; patterns could be “A”, “B”, “AB”, “BA”, etc.
  + If the user specifically provides comma-separated or whitespace-separated items (like numbers or words), we might consider patterns as sequences of those tokens. However, to keep scope simple, we assume character-based patterns. (A future enhancement could allow detecting repeated words or phrases by splitting on whitespace, but this is not in the initial scope unless explicitly needed.)
  + The app will preserve the input exactly as entered (including line breaks). It will not modify the input text except to highlight it in the output view.
* **Data Size Limits:** There is no hard-coded small limit on input size, but performance will naturally degrade with extremely large inputs. As a guideline:
  + The app should handle inputs of a few kilobytes (say up to 10,000 characters) relatively quickly on a modern device.
  + If the input is extremely large (e.g., 1 million characters), the algorithm may be slow or the browser may struggle with highlighting so much text. We will mention in documentation (if any) that it’s intended for moderate-size texts. Optionally, the app can warn the user if input is above a certain threshold (like “Warning: Large input may be slow to analyze”).
* **Pattern Detection Logic:** The core computation finds repeated substrings:
  + We define a parameter minLength (default 2) as the shortest length of pattern to consider. Single-character repeats can be very numerous and not always useful, but we might still allow it if user sets minLength = 1.
  + The algorithm will scan the input for all substrings of length >= minLength that appear at least twice.
  + We have to consider overlapping occurrences (e.g., input “ABA”, pattern “ABA” appears once, “A” appears twice (overlapping) if minLength=1). The algorithm will count overlapping occurrences as well.
  + There are various ways to implement this (brute force checking of all substrings, more efficient suffix tree/array or KMP-based search, etc.). The implementation detail: we can initially implement a simpler approach that is acceptable for moderate input sizes and optimize later if needed. The PRD doesn’t enforce which algorithm, but it requires that the result is correct (all repeats found) and reasonably fast on expected input sizes.
  + If case sensitivity is off, we will internally convert the input to one case (e.g., all lower-case) for pattern finding, but when displaying results, use the original-case text for highlights.
  + The result should avoid trivial outputs: for instance, if the entire input is one big repeat (like input is “XYZXYZ”), then “XYZ” is the main pattern. We might also list smaller sub-patterns like “X”, but perhaps de-prioritize them. We will show all found patterns or maybe have a cutoff to avoid an overwhelming list. For MVP, showing all is fine, but if input is very repetitive (like “AAAAAA”), the patterns list could be large (“AAAA” repeats, “AAA” repeats, “AA” repeats, “A” repeats). In such cases, perhaps limit to the longest pattern or most frequent patterns to avoid confusion. This is a design detail to be decided during implementation, but the requirement is to convey meaningful patterns to the user effectively.
* **No External Data:** The app does not fetch or send any data to external sources. The pattern detection is purely local. This means:
  + No API calls for analysis; all logic is in JavaScript running in the browser.
  + The user’s input stays in their browser. (If we had a download feature, say to save the results, that would generate a file on the client side.)
* **Local Storage Data:** As noted, we store some data in localStorage for persistence:
  + The raw input text may be stored. This is potentially large (in characters). We should ensure this is done carefully (e.g., compressing isn’t trivial for text in JS, but we could at least avoid doing it if input is huge).
  + Perhaps only store if below a certain size to avoid quota issues. Or we can slice it if too large.
  + Storing the patterns list is optional. Probably not needed because re-running detection is fine.
  + Settings (like minLength, etc.) are small and easy to store.
  + All stored data is keyed uniquely to this app (to avoid collisions). And we document how a user could clear it (clearing browser cache or clicking a “Clear saved data” if we provide a button for that).
* **Memory and Object Handling:** When the app highlights the input, it might create many DOM elements (for each occurrence). This is effectively a duplication of some data (the text is repeated in parts as content of spans). We should be cautious not to blow up memory:
  + We can limit highlights to patterns of interest (for example, maybe only highlight the top 10 longest or most frequent patterns if performance is an issue).
  + Or use an algorithm that finds non-overlapping occurrences for highlighting to reduce count, but that might miss some.
  + These decisions can be fine-tuned; the requirement is that the app should handle the typical cases smoothly and not crash on large input.
* **Data Integrity:** Ensure that the highlighting does not alter the actual text content. If a user were to copy the highlighted text, they ideally should get the original text without extra characters. Using semantic <mark> or <span> tags for highlight typically preserves copy-paste text, but needs verification. This is a nice-to-have to ensure the app doesn’t “corrupt” the data when copying out.
* **Security Considerations:** Since the input is user-provided, there’s minimal security risk (we’re not processing someone else’s data). However:
  + If the user inputs HTML, and we later render it (even inside a highlight span), we must escape any HTML tags to avoid inserting unintended HTML. For example, input <script> should not actually execute if present. We must treat the input as plain text. When rendering highlights, do not set innerHTML to raw user input without sanitizing or using text nodes. We likely will take the input and split it safely to insert highlight spans.
  + No other significant security issues since no external communication or persistent cookies beyond localStorage.

In summary, data handling is straightforward: accept text, analyze it locally, present results, and optionally save the text locally. The approach ensures user data is kept private and the functionality is achieved solely with client-side resources.

**Responsive Design**

The Persistent Pattern Detector will be built to be fully responsive, ensuring usability across a range of device sizes from large desktop screens to small mobile phones. Below are the key responsive design considerations and behaviors:

* **Layout Adjustments:**
  + On **Desktop/Large screens** (e.g., > 1024px width), we can utilize a multi-column layout. For instance, the Input Section could occupy the left column and the Results Section the right column. This allows the user to view the input and results side by side, which is convenient for cross-referencing patterns in context. A typical split might be 50/50 or 60/40 width depending on content.
  + On **Tablet/Mid-size screens** (~768px to 1024px), if space permits, side by side may still work, or a two-row layout (input on top, results below) might be better if side by side feels cramped. We will test breakpoints to decide when to switch.
  + On **Mobile/Small screens** (< 768px width, or particularly narrow devices), the layout will collapse to a single column stacked vertically:
    - The Input Section (text area and controls) will be on top so the user interacts with that first.
    - The Results Section will be below the input. After the user runs detection, they may need to scroll down to see the results.
    - We might add a slight tweak: after detection on mobile, we could auto-scroll the view to the top of the results section to reveal the patterns list to the user without them manually scrolling (since the input might have been covering the screen). This can be done via JS scroll and is a responsive behavior consideration.
* **Navigation and Controls on Mobile:**
  + Buttons (Detect, Clear) and settings toggles will likely be in a row or two rows above the text area on mobile. Ensure they are full-width or appropriately sized for easy tapping.
  + If multiple controls crowd the screen, consider making them collapsible or in an accordion. For example, a “Settings” accordion that the user can tap to reveal the min length slider and case toggle, saving space.
  + The patterns list on mobile might be long, but it will just naturally scroll. We should ensure the list items are not too wide (they will wrap text if needed).
* **Typography and Sizing:**
  + Use relative units (em, rem, percentages) for font sizes and container widths so that the UI scales with different screen densities and user zoom levels.
  + On small screens, maybe use a slightly smaller base font to fit more content if needed, but ensure readability (no text below ~14px).
  + The text area should resize (height can be a percentage of viewport or auto-growing). Perhaps on mobile, the text area shows fewer lines by default, but can scroll internally.
* **Touch vs Mouse Interactions:**
  + Hover effects are not available on touch, so anything that relies on hover (like tooltip hints for settings or highlighting on hover) should have an alternative. For example, if patterns list items had a hover tooltip showing the pattern in context, on mobile we’d need that information always visible or accessible via tap.
  + Use touch-friendly components: for example, if implementing a slider for min pattern length, ensure it’s easily operable via touch (or offer a number input as alternative).
  + The highlight tapping: if a user taps on a pattern in the list on mobile, it should bring that pattern’s occurrences into view. The smooth scroll is important here to move the text which might be off-screen into view.
* **Performance on Mobile:**
  + Mobile devices are less powerful, so our earlier performance considerations (like possibly using a Web Worker for analysis) become more important on small devices. We ensure that the UI doesn’t lock up while processing (especially since mobile browsers may warn or kill scripts that hog the main thread).
  + Animations on mobile should be kept minimal (or shorter durations) to maintain a snappy feel, as mobile devices also have different refresh characteristics.
* **Tested Breakpoints:**
  + We will define CSS breakpoints roughly around common sizes (e.g., 1200px for large desktops, 992px for small desktops/tablets, 768px for large phones, 576px for small phones, etc.). These will control major layout changes.
  + For example, at <768px, switch to single column. At >=768px, maybe two columns if space. At >=1200px, maybe widen margins or max-width so it doesn’t stretch too wide (very long lines of text are hard to read).
* **Flexible Components:**
  + The text input and results container should fluidly expand to fill space. On a wide screen, the text area might have a max-width for readability (no one wants a single line of text spanning 1000px). Similarly, the results list might be bounded.
  + Use CSS flexbox or grid to arrange the input and results sections, which can naturally collapse into a column on smaller screens by changing flex direction or grid template via media query.
* **Media Queries for Specific Tweaks:**
  + For instance, the pattern highlights might use slightly different styling on mobile if needed (like a slightly lighter color if the background is smaller).
  + The font used for code/monospace might be adjusted to a more standard system font on very small screens to improve legibility.
* **Orientation Changes:**
  + If the user rotates their phone or tablet, the app should smoothly reflow. If they had results shown, those remain and just reposition. Use window.onresize event if needed to adjust any JS-managed layout (though mostly CSS can handle it).
  + Ensure that if a user was scrolled in the text area and rotates, the scroll position remains relevant (this usually is fine).
* **Cross-Browser Consideration:**
  + Use prefix-free CSS where possible (modern browsers mostly handle standard flexbox, etc., but ensure compatibility).
  + Test on Safari iOS specifically for any quirks with textarea or scrolling within a div.
  + Use feature queries or graceful degradation: e.g., if CSS grid is not supported on an older browser, flexbox can be a fallback.

In summary, the design will be mobile-first (designing the small screen layout first, then enhancing for larger screens). This ensures that all functionality (entering data, clicking detect, viewing results) is achievable on a phone just as it is on a desktop, just with a different arrangement. The responsive behavior aims for ease of use and clarity of information on all devices.

**Performance Considerations**

To deliver a smooth user experience, especially given that all computation is on the client side, we must account for performance in both the algorithm and the UI rendering. Key performance-related requirements and strategies:

* **Efficient Pattern Detection Algorithm:**
  + The chosen algorithm for finding repeating patterns should be optimized for the expected input sizes. A naive double-loop search could become slow for long inputs (O(n^2) or worse). If inputs of significant length are expected, consider more efficient approaches (like suffix arrays, suffix trees, or hashing-based searches) which can bring complexity down (some advanced methods can find repeats in O(n) or O(n log n)).
  + We might implement a pragmatic approach: for instance, find patterns by gradually increasing length or using JavaScript’s built-in methods (like indexOf in a loop) which are often optimized in the engine. We ensure to test with large inputs to gauge performance.
  + Avoid extremely memory-heavy structures; for example, a suffix tree might use a lot of memory, which could be an issue in JS for very large input. Balance memory vs speed.
  + For initial version, it’s acceptable to handle moderate sizes well and perhaps note that extremely large inputs may take a while.
* **Non-Blocking UI (Use of Web Workers):**
  + Because the detection can be CPU-intensive, the app should avoid freezing the browser’s main thread during analysis. If the analysis on typical input (<10k chars) completes in under, say, 100ms, the user won’t notice much. But for larger inputs, consider performing the analysis in a **Web Worker** thread. Web Workers allow the heavy computation to run in the background:
    - The main page would send the input data to the worker (postMessage of the text or a transferable object if needed).
    - The worker runs the pattern detection and then sends back the results.
    - Meanwhile, the main UI can still respond (the spinner can animate, user could even cancel if we provided that option).
    - Using a worker adds complexity (especially transferring large data might have slight overhead and workers can’t access DOM), but it significantly improves perceived performance by keeping the UI responsive.
  + If using a worker isn’t implemented initially, ensure at least that we give feedback (loading spinner) and perhaps use setTimeout breaks in the algorithm for very large input to yield occasionally to the UI thread.
* **DOM Updates Optimization:**
  + Inserting highlights for patterns means manipulating the DOM (wrapping substrings with span elements). To optimize:
    - Do not update the DOM for each found occurrence iteratively in a loop, as that would cause repeated reflows. Instead, build an HTML string or document fragment that has all the highlights, and then inject it in one go into the DOM. This way, the browser will render the highlighted text in a single reflow.
    - Similarly, create the entire patterns list HTML as a fragment and append once, rather than appending each item one by one.
  + If the input text is very large and has many highlights, the number of DOM elements could be huge which can slow down rendering and even interactions (like scrolling). We may impose some limit or strategy:
    - For example, highlight only the first 1000 occurrences of any pattern and then maybe indicate “+more” if necessary (just an idea to avoid extreme cases).
    - Or after highlighting, measure performance; if it’s slow, consider using a Canvas or other approach for visualization in future (but that would sacrifice text copy ability, so likely stick to DOM text).
  + Use CSS classes for highlight styling rather than applying styles directly to each element, for efficiency and easier changes.
* **Caching and Re-computation:**
  + If the user doesn’t change the input or settings and hits “Detect” again, the app could recognize that it’s the same state and avoid recomputation (maybe just ignore the request or notify nothing changed). However, this is a minor optimization since users typically wouldn’t click detect twice without changes.
  + If we had a scenario of incremental input growth (like the user adds a bit more text and hits detect again), a smart algorithm could incrementally update results. This is complex to implement, so out-of-scope; we will simply recompute fully on each new request.
* **Memory Usage:**
  + Be mindful of using structures proportional to all possible substrings (which can be n^2 in worst case). That’s infeasible for large n. The algorithm must be clever or limited. We will likely not enumerate every substring explicitly for large input. Instead, focus on finding significant repeats.
  + JavaScript garbage collection should handle cleanup of large objects (like when we drop old results), but to assist it:
    - Set large unused arrays to null after use (e.g., if we allocate an array of length n for an algorithm, ensure it’s not kept around).
    - Avoid global variables that accumulate data inadvertently.
  + The localStorage usage is small unless input is huge; just be aware that storing a multi-MB string in localStorage could be slow (both writing and reading it). Perhaps we limit how often we save (maybe only on unload or when analysis is done, not on every keystroke).
* **Responsive Performance:**
  + Ensure acceptable performance on mid-range mobile devices. Test on a typical smartphone with a reasonably large input to see that it completes in a few seconds at most and UI doesn’t hang completely. If it’s slow, warn the user via a progress indicator so they know it’s working and not broken.
  + Use requestAnimationFrame for any visual updates that might be done in JS (like manual highlight drawing, if any).
* **Testing and Profiling:**
  + During development, use browser dev tools to profile the pattern detection on various input sizes. Optimize any hotspots identified (e.g., use efficient string methods, avoid unnecessary data copying).
  + Also test the memory profile for a large input to ensure we’re not leaking or holding onto large memory after clearing (ensuring cleanup code works).
  + Ensure that highlights rendering doesn’t overly degrade scroll performance. If a large text with many highlights becomes sluggish to scroll, consider simplifying the highlight (maybe only highlight on demand or use a virtualized list of text lines, which is complex; ideally avoid such extreme cases).
* **CDN/Bundle (Load Performance):**
  + The app itself should load quickly. It’s just static files (HTML, CSS, JS). We will keep the JS bundle lean, since we have no frameworks. Possibly a few KB of our own code plus any small libraries if used.
  + Use best practices: minify assets, use proper caching headers when deploying (though not a part of development spec, it’s a deployment config – just note that nothing heavy is loaded).
  + No large images or fonts are really needed; if a custom font for code is desired, we might include one, but that’s optional. Prefer system fonts to avoid extra downloads, for performance.
* **Performance vs. Accuracy Trade-off:**
  + If needed, we might decide to impose some limits to keep performance, such as ignoring extremely short patterns or extremely large inputs. But by default, we strive to accurately find all repeats in the given data.
  + The user should never experience the browser tab crashing or becoming unresponsive due to our app; if something is taking too long, we should handle it (maybe abort processing or at least inform the user).
  + In a worst-case scenario (very large input), we might implement a safeguard: e.g., if input length > some threshold, either refuse and ask to reduce, or do a partial analysis (like find only longest repeats). This can be documented if implemented.

Overall, the application should feel snappy for normal use cases. Through efficient coding and possibly multi-threading (Web Workers), we ensure that even computationally intensive tasks do not degrade the user experience. Rendering is optimized by batching DOM updates and keeping the interface lightweight. As a result, users can quickly get results and interact with the app without frustration.

**End of PRD**

This document provides a comprehensive blueprint of the Persistent Pattern Detector’s functionality and design from a development perspective. Developers can use these specifications to implement the front-end application in HTML, CSS, and JavaScript, confident that no further back-end or marketing input is needed. The focus is on delivering the described features with a clean, responsive UI and efficient client-side performance.