ChatGPT

Glazyr Chrome Extension – Product Requirements Document (PRD)

1. Overview & Goals

The Glazyr Chrome Extension is designed as the browser component of the Glazyr vision-first AI assistant. Its goal is to continuously "see" the user's web context and feed that information to Glazyr's AI core for understanding and action (following the **See → Understand → Act** paradigm [1]). In practice, the extension passively monitors the active Chrome tab (capturing URLs, DOM content, screenshots, user inputs, etc.) and actively provides a user interface for queries (e.g. "What is this page about?"). This extension bridges the browser ("eyes" and "hands") to the Glazyr orchestration engine ("brain"), in line with Glazyr's platform-agnostic architecture [2]. Its primary goal is to build a **vision-first automation orchestrator** for web browsing: capturing visual/DOM context, sending it to AI for analysis, and enabling AI-driven actions within the browser [3].

2. Functional Requirements

The extension's functionality follows Glazyr's core loop:

- **2.1 See (Context Capture)**

- **Page Detection & Navigation:** The extension must detect when the user opens or switches to a new tab or page (e.g. via `chrome.tabs.onUpdated` or `chrome.webNavigation.onCompleted`) [4] [5]. It should record the active tab's URL and title.
- **DOM Content Scraping:** Using injected content scripts, the extension must read page content. For example, it can execute `document.body.innerText`, extract metadata, links, and images, and send this text to the background script [6] [7].
- **User Input Capture:** Content scripts should listen for user interactions—clicks, keypresses, form inputs, and selections—and relay relevant details to the extension (via `chrome.runtime.sendMessage`) [6]. For instance, it might record form field values or button clicks to understand user intentions.
- **Network Context (Optional):** The background script can observe network requests (e.g. via `chrome.webRequest.onBeforeRequest`) on permitted hosts to gather subresource URLs or AJAX calls [8]. This helps Glazyr understand dynamic page content and third-party data.
- **Screenshots:** The extension should be able to capture a visual snapshot of the page (using `chrome.tabs.captureVisibleTab` with the `"tabs"` permission) when needed, supporting Glazyr's vision component [9].

- **Context Buffer:** The extension should maintain recent page contexts (in memory or `chrome.storage.local`) so Glazyr can reference prior pages in a session (e.g. "what did I look at 5 minutes ago?"). This aligns with Glazyr's design to loop through captured context [10].

- **2.2 Understand (AI Processing)**

- **Data Forwarding:** The extension must forward captured context (page text, screenshot, navigation history, user actions) to the Glazyr AI engine for analysis. This can be done either by calling a cloud AI API (via `fetch`) or by sending data to a local Model Context Protocol (MCP) server over WebSockets [11] [12].
- **Local/Remote Processing:** If a local MCP server is used (e.g. a Node.js process like YetiBrowser's MCP), the extension's background script will open a WebSocket to localhost and send structured commands (e.g. "browser_snapshot") with the page data [11]. Otherwise, the background can directly POST the data to a remote LLM endpoint (with CORS enabled) [12].

- **AI Response Handling:** The extension must receive AI-generated responses (summaries, answers, or action instructions) and pass them to the appropriate component. For example, if the AI returns a summary of the page, the extension should store it for quick access; if the AI returns a command ("click this button"), the background will coordinate its execution.

- **2.3 Act (Action Execution)**

- **User Query Interface:** The extension provides a UI (popup, side panel, or tab) where the user can ask Glazyr questions or issue commands. For example, clicking the toolbar icon opens a chat-like interface where the user can type queries about the current page [13].
- **Contextual Actions:** Based on AI output or user commands, the extension should perform actions on the page. Content scripts have the ability to click buttons, fill forms, or scroll to elements using `document.querySelector(...).click()` or form APIs [14]. For instance, if the AI says "subscribe to newsletter," the extension could find the email input and submit button and execute those actions.
- **Feedback and Highlighting:** When presenting answers, the UI can highlight relevant text or elements on the page (e.g. shading the paragraph that contains the answer) to provide visual feedback. This uses the DOM information the extension already captured.
- **Multi-turn Interaction:** The extension should support follow-up queries (multi-turn), allowing the user to refine or chain commands. The background must maintain dialogue state during a browsing session so that Glazyr can plan and execute multi-step workflows.

3. Non-Functional Requirements

- **Performance:** The extension must be lightweight and event-driven. It should only scrape or process content when needed (e.g. on page load or user request) to avoid slowing down browsing [5] [15]. Any AI calls (especially remote) should be asynchronous so the browser remains responsive.
- **Reliability:** The extension should robustly handle a variety of web content, including SPAs and dynamic pages. It must gracefully recover from errors (e.g. if a content script injection fails) and manage state even if the service-worker is suspended (per Manifest V3 constraints) [16].
- **Privacy & Security:** Captured data is user-sensitive. The extension must request and explain permissions clearly (see Section 6) and minimize data retention ("zero retention" of sensitive data) [17]. It should run entirely on-device where possible, and if using a server, ensure secure transmission. User data (pages, forms) should not be logged or stored long-term unless explicitly needed.

- **Cross-Browser Compatibility:** While targeting Chrome, the extension should use standard Chrome (Chromium) APIs so it can easily be ported to other Chromium-based browsers (Edge, Brave, etc.) [14] [18].
- **Usability:** Installation should be straightforward. On install, users will see permission prompts (e.g. "Read and change all your data on the websites you visit") [19]. The UI should be intuitive (e.g. chat interface in popup) and provide clear feedback (icon badge or notification when new context is captured) [20] [21].

4. Architecture and Components

- ### 4.1 Background Service Worker

- Acts as the central coordinator (the "brain").
- **Event Listeners:** Uses Chrome events to track browsing: e.g. `chrome.tabs.onActivated` and `chrome.webNavigation.onCompleted` to know which page is active [5]. It updates the active context and can decide when to scrape.
- **State Management:** Maintains session state (current and recent pages) in memory or `chrome.storage.local`. Keeps track of pending AI requests.
- **Messaging:** Receives messages from content scripts (page data, user actions) and from the UI. Dispatches messages to the UI or to external services.
- **AI Communication:** Forwards data to the AI logic. If using an MCP server, it opens a WebSocket to localhost (e.g. `ws://localhost:5000`) and sends commands like `browser_snapshot` with the page JSON [11]. If using cloud AI, it makes HTTP requests (`fetch`).

- **Action Execution:** Receives instructions from the AI (e.g. "click selector X") and uses `chrome.tabs.sendMessage` to tell content scripts to perform the action on the current page.

- ### 4.2 Content Scripts

- **Injection:** Specified in `manifest.json` under `content_scripts` with `"matches": ["<all_urls>"]` (or a restricted subset) [22]. In MV3 this can also be done at runtime with `chrome.scripting.executeScript`.
- **DOM Access:** Runs in the page context (isolated world) to read the DOM. For example, it might capture `document.body.innerText` for page text, extract metadata (title, headings, links), and send it to the background [7].
- **Event Handling:** Registers listeners on the page (e.g. `addEventListener('click')`, `input` events). When the user interacts (clicks, types, submits a form), the script can capture those events and relay details (e.g. which form was submitted) via `chrome.runtime.sendMessage` [6].
- **UI Injection (Optional):** It may insert small UI elements into the page if needed, such as an "Ask Glazyr" sidebar button or panel. If so, it should use an isolated shadow DOM to avoid CSS conflicts.

- **Security Note:** Content scripts cannot execute on Chrome's internal pages (e.g. `chrome://extensions`) and are subject to the page's CSP. They use only standard web APIs.

- **4.3 User Interface (UI)**

- **Popup/Side Panel:** The extension provides a chat interface for user queries. This can be a browser action popup (defined in `manifest.json`) or a side panel using Chrome's Side Panel API [13] [23]. It could also optionally open a full page (like a new tab) for more complex workflows.
- **Frontend App:** The UI can be built with a framework (e.g. React) and loaded in the popup. It communicates with the background via `chrome.runtime.sendMessage` or `chrome.storage`. For example, on opening, the UI might request the current page context [23].
- **Chat & Display:** The UI displays AI-generated answers, page summaries, or suggestions. It should allow text input for questions, and display results in a scrollable view. It may also present action buttons (e.g. "Click this link") if the AI suggests an interactive response.
- **Context Menu:** Optionally, the extension can add a context-menu item (on right-click) such as "Ask Glazyr about this" that sends selected text to Glazyr for immediate analysis [24].

- **Notifications/Indicators:** The toolbar icon can change (e.g. color or badge) when Glazyr is actively analyzing or when new insights are available [25].

- **4.4 Optional MCP Server / Native Components**

- **MCP Server:** For advanced use, Glazyr can employ a local MCP server (as in YetiBrowser). The extension's background would open a WebSocket to this localhost service and send structured commands. This allows heavy processing or streaming results from a local LLM, keeping data on-device [26] [12].
- **Native Messaging:** Alternatively, the extension could use Chrome's Native Messaging to connect to a native app or model. This would require the user to install a small helper program. It can enable full offline operation (embedding models) but adds installation complexity [27].
- **Fallback (Cloud):** If no local service is used, the background simply calls a remote AI API over HTTPS (with CORS), as a simpler architecture [12].

5. User Interaction Flows

- **Passive Tracking (Background Mode):** When the user browses normally, Glazyr runs quietly:
- The user navigates to a new page. The background script detects the URL change (`tabs.onUpdated` or `webNavigation.onCompleted`) [5].
- A content script (already injected or programmatically inserted) scrapes the page's text and DOM structure, and sends this data to the background [7].
- The background forwards the data to Glazyr's AI (local or cloud). The AI processes it (e.g. storing a summary of the page).
- Glazyr updates the UI in the background (for example, it might update the extension badge with a notification dot or a count of new insights) [25] [21].

- The user is unaware of this until they choose to interact; aside from the initial permission prompts and optional icon change, the extension is transparent.

- **Active Query (User-Initiated):** When the user wants answers or actions:

- The user clicks the Glazyr toolbar icon (or selects "Ask Glazyr" from context menu) [13]. The extension popup or side panel opens.

- The UI sends a message to the background requesting the latest context or directly includes the user's input. (If needed, the background may trigger a fresh scrape or use cached data.)
- The background sends the user's query and page context to the AI. (For example, "What is this page about?" along with the page text.)
- The AI returns an answer. The background relays this back to the UI. For example, the answer might be "This page is an article about X" or "The pie chart shows Y."
- The UI displays the answer in the chat window. If the answer includes an action (e.g. "click the subscribe button"), the UI can offer a button to execute it. When clicked, the background instructs the content script to perform the action. Alternatively, the extension may highlight relevant text on the page [28].

These flows allow both a **tracking mode** (passive, asynchronous analysis) and an **interactive mode** (on-demand Q&A). For example, one workflow is: on each page load, Glazyr captures and analyzes the page, and later the user can open the popup to see a summary or ask questions about what was captured [29].

6. Required Chrome APIs and Permissions

- `tabs` **API:** To detect tab activity, query URLs, and capture screenshots. (`tabs.onActivated`, `tabs.onUpdated`, `tabs.captureVisibleTab`) [4] [9].
- `webNavigation` **API:** As an alternative to `tabs`, to listen for when a page has finished loading (`webNavigation.onCompleted`) [5].
- `webRequest` **API:** To observe (but not block) network requests on pages (requires `"webRequest"` permission and appropriate host permissions). Note that in Manifest V3, `webRequest` can only *observe* or use declarative rules to block/modify requests [30] [31].
- `runtime` **/** `messaging`**:** For communication between background, content scripts, and UI (`chrome.runtime.sendMessage`, `onMessage`).
- `storage` **API:** To save any needed state or cache (e.g. recent page summaries) in `chrome.storage.local` [32].
- `activeTab` **permission:** Grants temporary access to the current tab when the user clicks the extension. Useful for actions like capturing a screenshot or injecting scripts on demand, with fewer prompts [33].
- **Host Permissions (**`<all_urls>`**):** Declared under `"host_permissions"` for MV3, or `"matches": ["<all_urls>"]` in content scripts. This allows the extension to run on any site to scrape DOM and observe requests [19] [33]. If true "read-any-site" access is needed, Chrome will show a warning ("Read and change all your data on the websites you visit") upon install [19]. It is best to minimize scopes (e.g. limit to necessary domains) or use `activeTab` to grant per-click permissions [34].
- **Other APIs:**
- `cookies`: If needed to capture session cookies (via `chrome.cookies`) for context [35].
- `sidePanel`: If using the experimental side panel UI (Chrome 93+).
- `notifications`: To alert the user (e.g. permission errors).
- `scripting`: For MV3 programmatic injection of scripts (if not using the declarative `content_scripts` manifest).

Each permission must be justified by functionality. For example, `tabs` is needed to capture the URL and screenshot, `storage` for saving state, `webRequest` for observing network context, and `<all_urls>`

so content scripts can run on any page. The extension should request broad permissions only if necessary [34] [33], and rely on `activeTab` for one-time access where possible.

7. Extension Capabilities

- **DOM Reading:** The extension can read the entire page's DOM and text content via content scripts [6] [7]. This includes visible text, hidden metadata (e.g. `<meta>` tags), and element attributes.
- **User Input Capture:** It can listen to and capture user inputs (keystrokes, form entries, button clicks) on the page via content scripts, even before the form is submitted [6].
- **Navigation Tracking:** Through `chrome.tabs` and `webNavigation`, it tracks URL changes, tab switches, and page loads [4].
- **Screenshots:** Using `tabs.captureVisibleTab`, it can take a snapshot of the current viewport (subject to permission) [9].
- **Network Observation:** With `webRequest` and host permissions, it can observe the URLs of all resources the page loads (scripts, XHR, images) [30].
- **Storage Access:** Via `chrome.storage` and content script access to `document.cookie` or `localStorage`, it can collect site-specific data if needed [35].
- **Automation Actions:** It can programmatically click or fill page elements using content script DOM APIs (essentially "click this selector" or `element.value = ...`) [14].
- **Logging and Debugging:** It can listen to console output (using `chrome.tabs.onUpdated` or devtools APIs) for diagnostics if needed, though this is more for development.
- **Cross-Browser Sync (optional):** If users sign in, it could sync certain settings or context via Chrome Sync (extension storage sync).

In short, Chrome extensions expose "virtually all high-level browsing activity" to Glazyr, constrained only by declared permissions [15]. This means Glazyr can have access to page text, user actions, cookies, and even capture visible tabs, fulfilling the agent's "eyes and hands" needs.

8. Manifest V3 Constraints

Under Chrome's Manifest V3 (MV3), several constraints must be considered:
- **Service Worker Background:** The background script runs as an event-driven service worker. It cannot stay alive persistently, so all state must be re-initialized on each event. Any long-running tasks must be broken into events. (However, using alarms or long-lived connections can work if kept alive by message passing.) [16]
- **Declarative Net Request:** Direct blocking or modification of network requests is no longer allowed except via static declarative rules. Glazyr's extension can *observe* requests with `chrome.webRequest` but cannot synchronously intercept them [31]. This mainly impacts if Glazyr ever tried to alter network calls; for passive context capture it's a non-issue.
- **Permission Prompts:** In MV3, additional prompts appear when powerful APIs or new hosts are used at runtime. For example, calling `tabs.captureVisibleTab` may require the `"activeTab"` or `"tabs"` permission and prompt the user [34]. Accessing a new domain via `webRequest` will also prompt for host permission [34]. Best practice: use `activeTab` for one-click actions and request the broadest (`<all_urls>`) only if truly needed.
- **Content Security:** MV3 enforces a strict CSP. Inline scripts and `eval()` are disallowed. All scripts must be in files; remote code loading is limited.

- **Side Panel API (MV3+):** The new side panel API is only available under MV3. Using it would require `"sidePanel"` permission and handling its event (the UI recommendations above reflect this).
- **Storage Quotas:** `chrome.storage.local` has quotas; large data (screenshots, long transcripts) may need careful management or offloading.

Glazyr's extension design must accommodate these. For instance, since the background may unload, it should save any important state to `chrome.storage` or only rely on immediate messaging. Also, any content script injection for dynamic pages should use `chrome.scripting.executeScript` (the MV3 method) if needed.

9. Optional Enhancements (Native/MCP Server)

- **Local MCP Server:** Glazyr can optionally use a local Model Context Protocol server (for example, YetiBrowser's architecture). In this design, a background Node.js process runs an LLM and context pipeline. The extension's background connects to it via WebSockets (e.g. `ws://localhost:port`) and sends serialized browser state [36] . This allows fully local processing (no data sent to cloud) and potentially better performance with a dedicated background process.
- **Native Messaging:** As an alternative, the extension could use Chrome's nativeMessaging to call a local executable (written in Python/C++). This gives Glazyr the ability to run native code (including local ML models) at the cost of requiring user installation of a host app [27] . This might be overkill unless absolute offline capability is needed.
- **Cloud-Only Mode:** If neither local server nor native host is used, the extension will simply call a remote AI API (this was already covered). This is simplest for users but relies on internet and a trusted server.
- **Existing Models:** The architecture can leverage open-source projects as references. For example, the Hangwin MCP Chrome extension and YetiBrowser use the pattern of extension + local server [37] . A Glazyr MVP might start with a direct cloud approach, but these options offer roadmap enhancements (for privacy or performance).

Overall, native/MCP approaches allow "local-first" operation (like BrowserOS) [37] , but they add complexity. The PRD assumes the extension should be fully functional without extra installations, so they are optional advanced paths.

10. Example Workflows

1. **Page Capture & Summarization:**
2. User visits *example.com/article*. The extension's background detects the new URL.
3. A content script immediately extracts the page's main text and sends it to the background (via `chrome.runtime.sendMessage`) [38] .
4. The background calls the AI (cloud or local) to generate a summary. For example, "This article is about the benefits of urban gardening."

5. The summary could be cached and also shown as a browser action badge or brief notification. If the user later opens Glazyr's popup, the summary appears instantly.

6. **Interactive Query:**

7. On *example.com/shopping*, the user clicks the Glazyr icon. The popup UI appears.
8. The user types “Find the cheapest price for the item on this page.” The UI sends the query to the background.
9. The background may send the query and page content to the AI. The AI responds with an answer (e.g. “The lowest price is $19.99 from Store X”) along with a suggestion “Click ‘Buy’ to purchase.”
10. The UI displays the answer. If the user clicks the suggested action, the extension’s content script clicks the “Buy” button on the page on behalf of the user. The page might navigate to a checkout.

11. **Form Autofill Command:**

12. The user goes to a login page. Using Glazyr’s popup, they ask “Fill in my login.”
13. The extension, knowing (from stored context) the user’s credentials, sends an action to the content script to populate the username and password fields and submit. This demonstrates the “Act” part (typing/clicking) using data from outside the page.

These workflows illustrate how the extension captures context, leverages AI, and returns results to the user via UI. All interactions are optional: the extension can run silently (tracking mode) and only present the UI when the user invokes it [39] [28].

References: This PRD follows Glazyr’s system vision [1] and is based on analyses of Chrome extension capabilities [40] [15] and feasible architectures [41] [26]. Key Chrome APIs and patterns (tabs, webNavigation, content scripts, service workers) are used as documented by Chrome [5] [33]. Existing projects (Hangwin MCP, YetiBrowser, BrowserOS) illustrate similar designs [37] [14].

---

[1] [2] [3] [10] [17] Glazyr_PRD.pdf
file://file_00000000fc1471fba0a1e0e5c80f1a51

[4] [5] [6] [7] [8] [9] [11] [12] [13] [14] [15] [16] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] Glazyr’s Required Data and Extension Feasibility (1).pdf
file://file_0000000072e871f5adfb6cee96619b4f