ChatGPT

SayPi Extension Authentication Flow PRD (Consolidated Revamp)

Document Information

- **Version**: 3.0 (Consolidated PRD + Addendum)
- **Date**: September 2025
- **Status**: Finalized Product Requirements
- **Supersedes**: Original Authentication Flow PRD v1.0 and Addendum v2.0

1. Problem Statement

The current authentication flow for the SayPi browser extension feels disjointed and unreliable. Users initiate sign-in from the extension's settings UI, are bounced through multiple tabs or windows, and often return to an unchanged settings view that still claims they are unauthenticated until they manually reload the page. This erodes user trust, obscures the success state, and leaves room for session inconsistencies and errors.

After completing the OAuth sign-in in a separate tab, the extension's settings page remains in an **unauthenticated** state (showing "Not signed in" with a Sign In button) because it has not been updated. This lack of immediate feedback causes users to think the sign-in failed, prompting confusion and frustration.

Only after the user manually refreshes or reopens the settings page does it update to show the **signed-in** state (displaying the user's name, usage quota, and a Sign Out option). This gap in feedback is a critical UX flaw: users do not see any confirmation of success and may abandon the process or attempt it repeatedly, undermining confidence in the extension's reliability.

2. Background & Context

- **Entry Points**: Users typically open the extension's settings UI from a chatbot host (e.g. via a "Voice Settings" button in a tool like Claude) or by clicking the browser extension's action icon. The sign-in process is initiated from this settings page.
- **Current Auth Mechanism**: Authentication relies on a session cookie (`auth_session`) set on **www.saypi.ai** after the user logs in via the web. The extension's background service worker detects this cookie and exchanges it for a JWT to authenticate API requests.
- **Extension MV3 Environment**: The settings UI is a React-based extension page (Manifest V3). When the user clicks "Sign In", it opens the OAuth login flow in a new browser tab (or window). After the user finishes logging in (via Google, GitHub, or email) and that tab closes, the settings page **does not automatically update** to reflect the authenticated state. The background script obtains the token, but the UI remains stale until a manual reload.

- **Recent Changes**: A recent Firefox-specific fix (to avoid background script self-messaging) ensures token refreshes succeed, but the core UX issue (lack of state update and multi-tab bounce) persists across Chrome, Chromium variants, and Firefox, on both desktop and mobile browsers.
- **Technical Limitations**: Mobile browsers impose additional constraints (e.g. Chrome on Android doesn't support the standard extension identity API) which make the current flow even more cumbersome on mobile devices.

3. Impact Assessment

- **User Confusion**: Without any post-login confirmation or UI change, users often assume the sign-in failed. This results in repeated attempts or abandoning the feature.
- **Support Load**: The confusing flow has led to increased support tickets and user feedback about "stuck" sign-in or "login not working," creating additional support burden.
- **Trust & Conversion**: Friction during the upgrade or sign-in process (especially when users hit a usage quota and need to sign in to proceed) reduces conversion to premium features. Currently only around ~30% of engaged users complete the sign-in process, indicating lost opportunities. The disjointed flow undermines trust at the critical moment users consider unlocking premium capabilities.
- **Mobile Friction**: On mobile platforms (Chrome Android, Firefox for Android), the lack of a smooth extension auth API leads to an even more fragmented flow (opening external browser windows or failing to redirect properly), further lowering sign-in success rates on mobile.
- **Security Perception**: The inconsistent state and reliance on hidden cookie exchanges can raise concerns that credentials or sessions are not handled securely. Users might worry their login didn't "take" or that something went wrong behind the scenes.

4. Goals & Desired Outcomes

1. **Seamless Sign-In UX** – Deliver a seamless, single-flow sign-in experience that clearly communicates progress and success or failure to the user at each step.
2. **Immediate State Sync** – Immediately reflect the authenticated state across all extension surfaces (settings page, pop-ups, content scripts) without requiring manual reloads after login.
3. **Best-Practice Implementation** – Align the authentication implementation with OAuth 2.1 best practices for browser extensions (using appropriate APIs for Chrome/Chromium and Firefox, both desktop and mobile).
4. **Strong Security Posture** – Maintain or improve the current security posture: no exposure of sensitive session cookies or tokens to unintended contexts, eliminate fragile workarounds, and support explicit user-initiated logout.
5. **Higher Conversion & Engagement** – Increase the sign-in conversion rate of engaged users from the current ~30% to **70%+**, by reducing friction and using progressive prompts to encourage sign-in at natural points (rather than forcing it upfront). This will directly support adoption of premium features and overall user retention.

Acceptance Criteria

- After authentication completes, the settings UI **automatically updates (within ~2 seconds)** to show the signed-in user's information (e.g. username, avatar, and premium status) **without requiring a manual reload**.

- While the authentication process is in progress (after user clicks "Sign In"), an in-flight state indicator (spinner and/or messaging) is visible, so the user knows something is happening.
- The end-to-end flow works on **Chrome/Chromium** (desktop and Android) and **Firefox** (desktop and Android). Where available, the extension uses sanctioned browser APIs (like identity APIs) for OAuth; on platforms where those are unavailable, the flow gracefully falls back to an alternative without confusing the user.
- Signing out (from the extension's UI) fully clears any stored tokens/credentials and immediately reverts the UI to a logged-out state. The user should get confirmation of sign-out and see the "Sign In" option again right away.
- Error conditions (e.g. user denies consent, network failure, session timeout) are communicated with clear error messages or dialogs. The UI offers a straightforward path to retry or to contact support if the problem persists.
- Unauthenticated users are allowed to perform a limited number of interactions (e.g. a few voice transcriptions) without forcing sign-in. After reaching the defined threshold or when attempting a premium feature, the extension triggers an appropriate sign-in prompt (toast or modal) with a clear value proposition, rather than simply failing silently.
- If a user's session expires or their token becomes invalid, the extension will attempt a **silent token refresh** in the background (if a refresh token or re-auth method is available) so that the user remains signed in without disruption. Users should only be prompted to log in again if the silent refresh fails or the session can't be restored.

5. Motivations & Non-Goals

Motivations: - Improve onboarding and build trust for first-time users, especially those coming via chatbot integrations where a smooth voice setup is critical to a good first impression. - Reduce engineering debt and flakiness by replacing the current custom cookie/token exchange logic with a more standard and robust OAuth flow. - Lay the groundwork for future premium feature adoption and entitlements which depend on a reliable authenticated state (e.g. subscription-based features, usage quota upgrades). - Increase conversion to premium plans by ensuring that the moment a user wants more (has high engagement or tries a premium feature), the sign-in process is easy and reinforces the value of signing in. - Address known pain points in the current system (multi-tab bouncing, lack of state sync, poor mobile support) to improve overall user satisfaction and trust in the extension.

Non-Goals: - This project will **not** redesign the broader settings UI or the extension's overall interface beyond the authentication-related components and prompts. The focus is strictly on the sign-in/up/out flow and related messaging. - We are **not** changing the backend identity provider or the core account management flows (e.g. no switching away from OAuth or changing the identity service). We leverage the existing OAuth provider (with possible minor backend enhancements for token exchange). - We are **not** implementing new payment or plan management flows here (though improving auth will indirectly support premium upgrades). - The project is not focused on chatbot UI changes except where they pertain to initiating or encouraging sign-in; any broader chatbot UX changes are out of scope.

6. Users & Stakeholders

- **End Users** – Individuals who install the SayPi extension for voice transcription or generation features. They benefit from a smoother sign-in that lets them unlock features like higher quotas or better voices with less hassle.

- **Product Support & Success Teams** – These teams will see reduced support tickets and frustration around login issues. A higher sign-in rate means more users move into the funnel for premium features, which aligns with success metrics.
- **Engineering Teams** – Specifically, the extension's background service worker team, frontend/pop-up team, and backend authentication service team. They will coordinate on implementing the new flow (including any needed backend changes). A cleaner auth flow also reduces maintenance of brittle code.
- **Product Management & Design** – Responsible for ensuring the authentication experience is consistent across surfaces. They will provide design for new prompts/modals and ensure messaging and UX meet user needs. They also define metrics of success for this auth revamp.
- **Security Team** – (Consultative) Ensures that the new OAuth implementation and token storage meet security requirements and that any new flows (especially on mobile or with refresh tokens) do not introduce vulnerabilities.

7. Requirements

Functional Requirements

1. **Unified Sign-In Flow**: Provide a cohesive sign-in entry point that does not confuse the user or leave orphaned tabs. Ideally, the sign-in should occur either in a modal window or a dedicated extension-controlled popup, rather than a full browser tab, to maintain continuity.
2. **Real-Time Auth State Broadcast**: The extension's background script must broadcast authentication state changes to all relevant extension contexts (settings page, popup, content scripts) in real time. This ensures that as soon as the user is authenticated (or logs out), every part of the extension can update its UI/state accordingly.
3. **Reactive UI Updates**: The settings UI (and any other relevant UI, such as a toolbar icon or chatbot overlay) should listen for auth updates via runtime messages or stored state changes, and automatically re-render to reflect the current auth status. No manual refreshes should be needed by the user.
4. **Multiple Auth Methods**: Support multiple authentication providers and methods: for example, OAuth with Google and GitHub (SSO), as well as email/password or magic link flows provided by SayPi's backend. The UI should accommodate these options (likely via the web login page).
5. **Return to Origin Context**: If the user initiates login from a particular context (for instance, from a chatbot conversation where they clicked "Voice Settings"), the flow should preserve that context. After successful sign-in, the extension should offer a quick way to return to the original tab or state (e.g., automatically focusing the chatbot tab or providing a link back).
6. **Progressive Engagement (Deferred Auth)**: Implement a **progressive authentication prompt strategy** instead of forcing sign-in immediately. Unauthenticated users should be able to use the extension in a limited capacity (e.g. a few voice transcriptions) to experience its value. The extension will then present **graduated prompts**:
7. A gentle, non-blocking prompt (such as a toast notification) after a certain number of uses, highlighting benefits of signing in (e.g. "Save your preferences by creating a free account").
8. A stronger prompt (modal dialog or sidebar) after further engagement, showing a comparison of free vs. premium features and urging sign-in.
9. A hard requirement when a user tries to access a **premium feature** (e.g., higher-quality voices or extended quota), explaining that sign-in is required to proceed.

10. **Silent Token Refresh**: The system should maintain user sessions seamlessly. Implement background token refresh logic so that as long as a user remains active (or has granted offline access), their authentication stays valid without frequent re-prompts. This includes using refresh tokens or similar mechanisms provided by the backend to obtain new JWTs before the old ones expire, all without user intervention (unless a refresh fails or requires re-auth).

Security Requirements

1. **OAuth Authorization Code Flow with PKCE**: Use the OAuth 2.1 Authorization Code flow with Proof Key for Code Exchange (PKCE) for all sign-in operations. Where possible, leverage the browser's extension identity APIs (`chrome.identity.launchWebAuthFlow` in Chrome/Edge, `browser.identity.launchWebAuthFlow` in Firefox) to handle the OAuth process securely. This ensures no credentials or tokens are exposed to the client-side without proper exchange.
2. **No Plaintext Credentials or Cookies**: Never expose the session cookie or user credentials directly to the extension frontend. The extension should store only the resulting tokens (access token/JWT, and refresh token if applicable) in its secure storage. The previous workaround of posting the `auth_session` cookie via an HTTP request should be eliminated [1].
3. **Secure Token Storage**: Tokens must be stored in extension-controlled storage (e.g. `chrome.storage.session` for in-memory and `chrome.storage.local` for backup) and, if persistent storage is needed, should be encrypted at rest. The extension should avoid using any web accessible storage for tokens. On logout, all tokens and related credentials should be cleared promptly.
4. **State and Nonce Validation**: Every OAuth callback must be validated by verifying the `state` (and nonce, if used) matches the one the extension initiated. This prevents CSRF and malicious callbacks. Use cryptographically secure random values (`window.crypto.getRandomValues`) when generating state/nonce to ensure they can't be guessed.
5. **Explicit Logout Support**: Provide a clear way for users to log out. The logout process should revoke tokens on the server (if an endpoint is available for token revocation) and clear all local auth state (tokens, cookies) from the extension. After logout, the extension should return to an unauthenticated state and prompt the user to sign in again if needed.
6. **Token Refresh & Rotation**: If refresh tokens are used, implement refresh token rotation and revocation as recommended by OAuth 2.1. Each time a refresh token is used, the backend should issue a new one and invalidate the old. The extension must handle this (store the new token, discard the old) and be prepared to detect if a refresh token was revoked or reused (which could indicate a breach, prompting a full re-auth).
7. **Minimal Permissions & Secure Calls**: Use only HTTPS for all authentication-related network calls. Ensure that any extension permissions requested (like access to identity APIs or webRequest for intercepting callbacks on Firefox) are as limited as possible to achieve the flow, to minimize security attack surface.

User Experience (UX) Requirements

1. **Progress Indicators**: Provide clear feedback during the sign-in process. For example, when the user clicks "Sign In", the UI should immediately show a loading spinner or message ("Signing you in…") so the user knows the process has started. Likewise, during token refresh in the background, if it impacts the UI (e.g., content temporarily locked), some indication should be given.
2. **Auto-close & Return**: If the sign-in flow opens a new window or tab (for example, an OAuth login page), it should automatically close upon completion of the authentication (once tokens are

acquired). The user should be routed back to where they started. For instance, if they began in the extension settings page, focus that page; if they started from a chatbot overlay, return them to that context with a confirmation message. A transient confirmation (toast notification like “ You’re now signed in!”) should inform them the process succeeded.

3. **Error Handling UI**: For any errors during auth (e.g. user closed the login window, network issues, permission denied), present an informative message. This could be inline on the settings page or a modal. Offer actionable choices: e.g., a “Retry” button for transient errors, or a “Contact support” / “Troubleshoot” link if the issue is likely to persist. The user should never be left wondering what to do next if sign-in fails.
4. **Accessibility**: Ensure the sign-in flow is accessible. This includes proper focus management (e.g., if a modal opens, focus on it), screen reader announcements for status changes (like “Signing in…” live region), and keyboard-only navigation support. Visual cues (like focus outlines) should be preserved. All new UI (prompts, toasts, modals) should be tested with accessibility in mind.
5. **Trust & Transparency**: Incorporate visual trust indicators into the authentication UI to reassure users that the process is secure. For example, display the logo of the auth provider (Google, GitHub, etc.) on the sign-in button or modal, show a “Secured by [Provider]” subtitle or badge, and include a short privacy statement like “We never see your passwords – you’re securely authenticating with the provider.” The sign-in screen can also mention the expected duration (“Usually takes less than 30 seconds”) to set user expectations. Use the provider’s brand colors and design conventions for familiarity and trust. These elements will help users feel comfortable proceeding with the login.

8. Progressive Authentication & Engagement Strategy

One major enhancement in this revamp is a **progressive authentication strategy**. Instead of immediately blocking features until a user signs in, the extension will gradually encourage authentication as the user becomes more engaged, and only enforce sign-in when truly necessary (e.g. premium features). This aims to provide value upfront to free users, then convert them to signed-in (and eventually premium) users through timely prompts.

8.1 Tiered Authentication Prompts

We define three stages of user engagement for the extension’s voice features:

- **Free Tier (Unauthenticated)** – New users who have not signed in can use basic functionality for a short period. For example, allow up to **5 voice transcriptions** or interactions without requiring sign-in. During this phase, the extension will display small, non-blocking notifications to highlight benefits of signing in, but it will not interrupt the user’s first few experiences. We will also silently track usage metrics in this state.
- **Engaged User (Prompted Sign-In)** – After a user has demonstrated interest (e.g. after their 5th successful transcription), the extension will trigger progressive prompts:
- After the 5th interaction: show a gentle **toast notification** saying something like “☆ Love SayPi? Create a free account to save your voice settings and get more features.” This prompt does not block usage; it can disappear after a few seconds if ignored.
- After the 10th interaction (or another threshold): show a slightly more prominent **soft modal** or in-line prompt within the settings UI or chatbot interface. This modal can outline what additional benefits they unlock by signing in (for instance, saving preferences cloud-sync, more usage quota,

etc.) and provide a one-click "Sign In to Continue" button. The user can still dismiss this modal if they are not ready.

- **Premium User Gate (Authenticated Required)** – When the user tries to access a feature that is classified as **premium** (e.g. uses an enhanced voice that requires a subscription, or exceeds the free quota of usage), the extension will **require** authentication at that point. This is a hard gate: a modal appears explaining that a sign-in is required to proceed (with messaging highlighting the value of premium features). The user must sign in (and possibly upgrade if needed) to continue past this point.

Premium features that will explicitly trigger a mandatory sign-in include: - Using enhanced voice models (e.g. ElevenLabs voices or other premium voice engines). - Exceeding the free transcription quota (for example, after 5 free uses, additional uses require login). - Cross-device sync or advanced settings persistence (only available to signed-in users). - Any feature marked as for premium-tier users.

By structuring the flow this way, new users can immediately experience the core functionality (voice transcription) without barriers, increasing the likelihood they see value. As they engage more, we progressively educate them on the benefits of signing in, and by the time they hit a limit or premium feature, they are primed to log in (and potentially later upgrade to a paid plan). The goal is to maximize the conversion of engaged free users into authenticated (and eventually paying) users, thereby hitting the 70%+ sign-in rate target.

Implementation Details for Progressive Prompts

- The extension will maintain counters in memory (and possibly in local storage if needed) for each user session to track usage events (e.g. how many transcriptions have been done).
- **Engagement Triggers**: The logic will trigger prompts based on these counters:
- After the 5th transcription (and maybe again at a certain count like 10), if the user is still unauthenticated, trigger the defined UI prompt.
- When the user hits a premium feature action, trigger the blocking sign-in modal.
- Each prompt type (toast vs modal) will only be shown once per user per session threshold to avoid spamming. If the user dismisses it, we might increase a dismissal count and perhaps show a reminder again after a few more interactions, but carefully so as not to annoy.
- These triggers and thresholds can be A/B tested (see **Analytics & Experimentation** section) to optimize conversion. For example, we might experiment with showing the first prompt after 3 uses vs 5 uses to see which yields better sign-in uptake.

For clarity, a pseudo-code representation of how the extension might decide which prompt to show can be as follows:

```
interface AuthPromptContext {
  trigger: 'quota_limit' | 'premium_feature' | 'engagement_threshold' |
'settings_save';
  userState: {
    sessionsCount: number;
    lastPromptShown: Date;
    dismissalCount: number;
  };
  presentation: 'toast' | 'inline' | 'modal' | 'sidebar';
}
```

In practice: - `trigger` indicates what caused the prompt (hitting a free quota limit, attempting a premium feature, crossing an engagement threshold, or perhaps trying to save settings to cloud). - `userState` keeps info about how many sessions or interactions the user has had, when the last prompt was shown and how many times they dismissed prompts, to modulate frequency. - `presentation` defines the UI style for the prompt (small toast, embedded inline message, modal dialog, or perhaps a slide-out sidebar prompt in the chatbot UI).

Using this context, the background or content script can decide: e.g., if `trigger === 'quota_limit'` or `'premium_feature'`, use a modal since it's a required action; if `trigger === 'engagement_threshold'` and `dismissalCount` is low, maybe use a gentler toast or inline message.

All these prompts will funnel the user into the primary authentication flow when they choose to sign in (see next sections), so the actual sign-in mechanism remains the same – we are just changing **when and how we ask** the user to log in.

8.2 Contextual Sign-In Entry Points

In addition to automatic triggers, we will ensure there are contextual entry points for sign-in throughout the UI: - The settings page will continue to have a prominent "Sign In" button when the user is not signed in. - If the user tries to use a premium setting (like selecting a premium voice in a dropdown), we can intercept that action and show a prompt to sign in to enable that feature. - We may also introduce a small sign-in callout in the chatbot overlay itself (for example, a notice like “You’re using SayPi in free mode. Sign in to unlock more.”) that can be dismissed. This addresses the earlier **Open Question** about exposing auth status in chatbot overlays – the plan is to show a minimal status or prompt in context to encourage sign-in without requiring the user to open the settings page first.

Overall, the progressive strategy should feel like a natural extension of using the product: the user is guided towards signing in at the point it becomes relevant to them, rather than being forced through a login wall at the very start.

9. Design Options for Technical Implementation

To improve the authentication flow, several technical approaches were considered:

Option A – OAuth via Extension Identity API (Recommended)

Leverage the browser extensions’ identity APIs to perform the OAuth flow. In Chrome (and Chromium-based browsers), use `chrome.identity.launchWebAuthFlow`; in Firefox, use `browser.identity.launchWebAuthFlow`. This API opens a controlled popup window for OAuth and can be configured to follow redirects to a special extension URI.

- **Flow**: The extension calls the identity API with the OAuth authorization URL (including PKCE parameters and a redirect back to an extension-provided URI). The browser opens a small popup window (or an in-app browser view on mobile, if supported) for the user to complete login. Upon successful authentication, the OAuth provider redirects to the extension's redirect URI (which the identity API monitors), providing an authorization code. The identity API then gives this code back to the extension.

- The extension's background script exchanges the authorization code for tokens (JWT access token and a refresh token if provided) by calling a backend token endpoint. On success, the background stores the tokens and notifies extension UIs of the new auth state.
- **Pros**: This aligns with OAuth best practices and uses official APIs designed for auth flows. It avoids full-page redirects and tab juggling, instead using a focused modal/popup. It provides built-in redirect handling and state management. Security is improved through PKCE and controlled redirects. It also works on desktop for Chrome and Firefox, and on some mobile variants that support the identity API.
- **Cons**: Requires support from the backend for the Authorization Code + PKCE flow (including handling the code exchange and issuing refresh tokens). We need to register and whitelist the extension's redirect URI with the OAuth provider. On mobile Chrome for Android, `chrome.identity` might not be fully supported (Chrome Android doesn't support extension popups in the same way), so this approach may not work on all platforms — we will need fallbacks for those cases.

Option B – Embedded Tab with Message Passing (Fallback Approach)

Improve the current approach by still using a regular browser tab for OAuth, but adding mechanisms to communicate back to the extension and close the tab automatically upon completion.

- **Flow**: The extension opens the OAuth login page in a new tab as it does currently. However, after the user signs in, the final redirect goes to a special page (hosted or extension page) that contains a script to call `chrome.runtime.sendMessage` (or `browser.runtime.sendMessage` in Firefox) to the extension with the result (the auth code or token). That page can then immediately close itself (via a `window.close()` or similar, if permitted).
- The extension listens for the message from that tab, retrieves the code/token, and proceeds to exchange it for a JWT as in Option A. The settings UI, upon receiving the message (or seeing the new token in storage), updates to the signed-in state.
- **Pros**: This option requires less new infrastructure. It can reuse existing backend endpoints (if we were doing an implicit flow or a cookie-based flow, it could even use that). It's easier to implement quickly since it's closer to what we have, just adding message passing and auto-close. No special API support needed.
- **Cons**: The user still experiences a full new tab or window for the login, which feels disjointed. On mobile, switching apps or tabs is clunkier (and `window.close()` might not work in all mobile scenarios, leaving a blank tab open). Security is weaker compared to Option A; we'd likely still rely on the presence of a session cookie, which is less ideal. It's essentially a patch on the current flow, and while we'll use it as a **temporary fallback**, it's not the desired long-term solution.

Option C – In-Extension Embedded WebView/IFrame

Embed the login page directly into the extension's settings UI (for example, in a modal or a dedicated section of the settings page) using an `<iframe>` or Chrome's `webview` tag.

- **Flow**: When the user clicks "Sign In", instead of opening an external page, the extension could display the OAuth login form right within the extension UI.
- **Pros**: This provides a single-surface experience; the user never leaves the settings page. It could theoretically be a smoother visual experience if it worked.
- **Cons**: **Significant issues**: Many OAuth providers (Google, GitHub, etc.) explicitly block their login pages from being loaded in iframes or webviews for security reasons (to prevent clickjacking and phishing). The extension would also have to relax Content Security Policy (CSP) to allow the OAuth domain, which is risky. Handling redirects within an iframe is complex. This approach may also violate Chrome Web Store policies or just outright fail due to provider restrictions. Due to these issues, this option is not considered viable.

Decision: Option A is the **recommended approach** for desktop environments given its security and UX benefits, with Option B implemented as a fallback for environments where Option A is not fully supported (e.g., certain mobile scenarios or in interim if backend support for Option A is incomplete). Option C is not pursued due to security and feasibility concerns.

10. Recommended Approach

We will proceed with implementing **Option A (OAuth with Identity API + PKCE)** as the primary flow, and use an improved **Option B (tab-based flow with messaging)** as a fallback when needed. Additionally, we will integrate the progressive sign-in prompting strategy into this implementation. The approach consists of several key components:

1. **OAuth Code Flow via Identity API** – Implement the Authorization Code flow with PKCE for Chrome and Firefox extensions:
2. Use `chrome.identity.launchWebAuthFlow` / `browser.identity.launchWebAuthFlow` to initiate the OAuth login in a modal/popup.
3. Handle the redirect back to the extension (using `chrome.identity.getRedirectURL()` to generate an allowed callback URL).
4. When the identity API returns the authorization code, exchange it on the backend for a JWT access token (and refresh token if provided).
5. Ensure the OAuth `state` is verified during the callback to prevent any CSRF attacks.
6. **Unified Auth Controller (Background Script)** – Create a centralized module in the background service worker to manage authentication logic. This module will:
7. Initiate the login flow (abstracting whether it uses the identity API or fallback method depending on platform capabilities).
8. Handle the exchange of authorization code for tokens by calling the appropriate backend token endpoint.
9. Securely store the obtained tokens (in memory and/or encrypted storage as needed).
10. Broadcast auth status updates to all extension parts (send a standardized message like `AUTH_STATUS_UPDATED` with user info on success, or `AUTH_ERROR` on failure).
11. **Real-Time UI Update & Feedback** – Update the extension UIs to respond to auth events:

12. When login begins, immediately show an **optimistic loading state** (spinner or dimmed UI with a message) in the settings page (and any other relevant UI) indicating that sign-in is in progress.
13. Listen for the auth status messages or changes in a shared auth state, and when a “signed in” event is received, hide the loader and update the UI to show the user’s information (e.g., display username, profile pic, etc., in the settings page).
14. Show a success toast notification once the user is signed in (e.g., “You’re now signed in!”). In case of failure, show an error notification or inline message (with guidance to retry).
15. **Return-to-Context & Closing Flows** – Implement return-flow tracking so that after authentication:
16. If the user came from a chatbot tab or any specific content context, we refocus that tab or provide an easy link back. (This can be done by storing the originating tab ID or URL before starting auth, then on success, using `chrome.tabs.update` to refocus it or sending a message to that tab’s content script to show a confirmation).
17. The intermediate login window/tab closes itself automatically once the process is done (for identity API, this happens by design; for fallback flow, we’ll ensure the final page calls `window.close()` after notifying the extension).
18. **Fallback and Compatibility** – In environments where the identity API is unavailable or insufficient (e.g., Chrome on Android, or if the user’s browser blocks the popup), the auth controller will detect this and transparently switch to the fallback (Option B) flow:
19. For Chrome Android, this might mean opening the OAuth URL in a Chrome Custom Tab and intercepting the callback via a custom scheme.
20. For Firefox Android, it could mean opening a new tab and using the webRequest API to catch the redirect.
21. In any case, the user experience is kept as seamless as possible (auto-close tabs, show loading indicators, etc. as in the main flow).
22. **Progressive Prompt Integration** – Introduce the **progressive sign-in prompt system** into the extension:
23. Build the logic for tracking usage and triggering the **tiered prompts** described in section 8. This will likely live in the background script or in a content script that monitors usage events.
24. Ensure that when a user chooses to sign in from any of these prompts (toast, modal, etc.), it invokes the same unified auth controller flow outlined above.
25. The prompts themselves will be implemented with reusable components (so a toast vs modal are just different UIs calling into the same auth start function).
26. This step ties the product strategy into the technical implementation, ensuring that increased conversion goals are met alongside the improved UX flow.
27. **Telemetry & Cleanup** – Remove the old cookie-based auth hack once the new flow is confirmed working:
28. The current system where the extension sometimes posts the `auth_session` cookie in a JSON body to retrieve a token (a fragile fallback) will be deprecated.
29. We will keep it only as long as needed behind a feature flag for safety. With the identity API approach and proper token storage in place, this should no longer be necessary. We will monitor the new flow’s success via telemetry to ensure parity (e.g., if any percentage of users fall back to old method, investigate and address).

This recommended approach ensures we tackle both the **user experience** (with a smoother flow and progressive engagement) and the **technical reliability/security** (with a proper OAuth implementation). Next, we detail specific aspects of the implementation, including OAuth details, mobile handling, state management, and more.

11. Enhanced OAuth Implementation

This section describes the technical specifics of the OAuth 2.1 Authorization Code with PKCE implementation in the extension, which is the core of the new sign-in flow.

11.1 Primary OAuth Flow (Identity API with PKCE)

When starting the auth flow via Option A (the default on desktop browsers):

- **Authorization Request Construction**: The extension will construct the OAuth authorization URL with all required parameters. For example, in pseudo-code:

```
const authorizationUrl = new URL(`${config.authServerUrl}/oauth/authorize`);
authorizationUrl.searchParams.set('client_id', config.clientId);
authorizationUrl.searchParams.set('response_type', 'code');
authorizationUrl.searchParams.set('redirect_uri',
chrome.identity.getRedirectURL());
authorizationUrl.searchParams.set('scope', 'profile voice_generation
offline_access');
authorizationUrl.searchParams.set('code_challenge', generatedCodeChallenge);
authorizationUrl.searchParams.set('code_challenge_method', 'S256');
authorizationUrl.searchParams.set('state', generateRandomState());
authorizationUrl.searchParams.set('prompt', 'select_account');
```

Here, `chrome.identity.getRedirectURL()` provides a redirect URI that the extension is registered to handle (it usually looks like `https://<extension-id>.chromiumapp.org/...`). We include `offline_access` in the scope to get a refresh token if the provider supports it. We also include a `prompt=select_account` to ensure the user can switch accounts if they want (this forces the OAuth provider to show an account chooser every time rather than auto-continue with an existing session).

- **Launching the Flow**: We call `chrome.identity.launchWebAuthFlow({ url: authorizationUrl.toString(), interactive: true }, callback)`. This opens the OAuth URL in a small popup window (for desktop Chrome) or in the appropriate UI. On Firefox, a similar call via `browser.identity.launchWebAuthFlow` will be used.
- **OAuth Callback Handling**: The OAuth server, after user login, will redirect to our extension redirect URI with a `code` and the `state`. The identity API intercepts that redirect. In the callback of `launchWebAuthFlow`, we will get the final redirect URL. The extension can parse out the `?code=` parameter from that URL.
- **State Verification**: We will have stored the `state` we sent (e.g., in a closure or a global variable) when initiating the flow. We must compare it with the `state` returned in the redirect URL to ensure they match. If not, we abort (this would indicate a potential malicious redirect).
- **Token Exchange**: Assuming state is valid, the extension's background script then makes a secure request to the backend's token exchange endpoint (e.g., `POST https://api.saypi.ai/oauth/token`) with the following:
- `grant_type=authorization_code`
- `code=<the authorization code>`

- `redirect_uri=<the same one used above>`
- `code_verifier=<the original random verifier corresponding to the code_challenge>`
- The backend (or OAuth provider) responds with a JSON containing the `access_token` (JWT) and possibly a `refresh_token`, along with expiration info.
- **Token Storage**: The background script stores the `access_token` (and refresh token) in extension storage. Specifically, we plan to use `chrome.storage.session` for primary storage since it's in-memory and cleared on browser restart (for security), and also store an encrypted copy in `chrome.storage.local` as a backup so the user doesn't have to log in again after a browser restart (unless we intentionally want that for security – but since we have refresh tokens, we can persist encrypted refresh tokens).
- **Auth Broadcast**: After successfully obtaining tokens, the background script will broadcast an `AUTH_STATUS_UPDATED` message (e.g., via `chrome.runtime.sendMessage` and possibly via `chrome.tabs.sendMessage` to any open content scripts) containing the essential user info (which it can get from decoding the JWT or from a userinfo API call if needed). It will also update any centralized auth state (see section on State Management).
- **Fallback to Interactive**: If any step fails (e.g., the identity API returns an error or the user closed the popup), the extension catches that and can retry or fall back. For instance, if `launchWebAuthFlow` is not supported (on some platform) or fails to retrieve a code, we might resort to Option B's approach.

By using this approach, we ensure that the extension follows OAuth best practices. The tokens are obtained through a secure server exchange (Authorization Code grant) rather than exposed directly to the front-end via an implicit flow. PKCE (Proof Key for Code Exchange) adds an extra layer of security to prevent interception of the auth code. The use of the browser's identity API means the browser itself helps manage the redirect and ensures the redirect URI is valid for this extension, reducing the chance of leakage.

11.2 Silent Token Refresh Strategy

A crucial part of maintaining a seamless experience is refreshing auth tokens in the background before they expire, so users remain authenticated without unexpected interruptions. Our strategy for silent refresh is as follows:

- **Token Lifetime**: Suppose the JWT access token has a validity of 1 hour. We will schedule a refresh slightly before it expires (to account for network delays and to avoid the token fully expiring).
- **Using Chrome Alarms (MV3)**: In a Manifest V3 extension, background pages are service workers that can be inactive when nothing is happening. We will use `chrome.alarms` to schedule a wake-up call for token refresh. For example, if a token expires at 14:00, we might set an alarm for 13:55.
- **Storing Refresh Metadata**: When we store the tokens, we also store their expiration time (decoded from JWT or provided). We schedule an alarm (or a timeout if the service worker stays alive) for a few minutes before expiry (e.g., 5 minutes before).
- **Refresh Flow**: When the alarm triggers, the background script will attempt a refresh:
- If we have a refresh token: call the backend token endpoint with `grant_type=refresh_token` and present the refresh token. If accepted, we get a new access token (and possibly a new refresh token).
- If no refresh token is provided by our system (e.g. if backend opted for short-lived access tokens without refresh tokens), we might attempt a silent re-auth. For Google, for instance,

`launchWebAuthFlow` can be called with `interactive: false` and it may return a fresh token if the user's session is still valid. But since we control our own auth, we likely will have refresh tokens.

- **Retry and Backoff**: Implement retry logic for refresh in case of transient failures:
- If a refresh attempt fails due to network or a 500 error, try again in a few seconds. Possibly use an exponential backoff (e.g., wait 1s, then 2s, then 4s) for a couple of attempts.
- If refresh fails due to token invalid (e.g., refresh token expired or revoked), then treat it as needing user intervention (we will then consider the user logged out and prompt them to sign in again when they next interact).
- **Limit Silent Attempts**: We won't retry indefinitely. For example, after 3 failed refresh attempts in a row, we stop trying silently and mark the user as logged out (or prompt them that they need to sign in again).
- **Refresh Token Rotation**: We will follow modern OAuth 2.1 guidance for refresh token security:
- Every time we use a refresh token, if the backend issues a new refresh token, we'll update our stored token with the new one and replace the old (so the old one can be invalidated server-side).
- By doing this, if a refresh token ever leaks, the window of misuse is reduced since using a rotated-out token should invalidate the whole session.
- We will store a token family identifier if provided (for detecting token reuse as a possible sign of theft).
- **Memory vs Storage**: Use `chrome.storage.session` for the access token so that it's not written to disk. For the refresh token, since we want persistence across browser restarts, we'll store it in `chrome.storage.local` but likely encrypted. The encryption key could be derived from a combination of extension-specific info and maybe user credentials (though we don't want to store those). Even if not strongly encrypted, storing in local (which is not easily accessible to the outside world for extensions) is an improvement over cookies in terms of scoping.
- **Cross-Context Consideration**: All this happens in background. The content scripts or UI should ideally not need to know or care about refreshes happening. They will simply always query a central AuthState which remains "authenticated" because the background updates the token in time. If, however, the user happens to open the settings UI exactly when a refresh is happening, we might show a brief spinner or disabled state for premium features until the new token arrives (to handle the case of a brief lapse).

By implementing silent refresh, we meet the goal that a returning user (say after a day or two, within token refresh period) does not have to sign in again—they should find themselves still signed in, or seamlessly re-signed-in without any prompt. Our success criterion is to have >99% of token refreshes happen without user noticing or needing to do anything.

12. Mobile-Specific Implementation Considerations

Supporting mobile browsers (Chrome on Android, Firefox on Android) is crucial, as a subset of our users use voice features on mobile or tablet. These platforms sometimes lack full support for extension APIs like the identity API, so we need platform-specific handling.

12.1 Platform Detection and Routing

We will implement a strategy pattern for auth flows based on platform capabilities. For example:

```
interface PlatformAuthStrategy {
  platform: 'chrome-desktop' | 'chrome-android' | 'firefox-desktop' | 'firefox-
android';
  method: 'identity-api' | 'custom-tabs' | 'webview-intercept';
  capabilities: {
    silentRefresh: boolean;
    biometricAuth: boolean;
    secureStorage: boolean;
  };
}
```

This is a conceptual representation; essentially, upon initialization, the extension can detect: - Browser name and version (via `navigator.userAgent` or extension APIs). - Whether `chrome.identity` exists and works. - Whether certain APIs (like `chrome.tabs` or `browser.webRequest`) are available (especially on mobile).

We then choose a method: - For **Chrome Desktop** and **Edge/Desktop Chromium**: use `'identity-api'` since it's available and reliable. - For **Firefox Desktop**: use `'identity-api'` (Firefox supports a similar API). - For **Chrome on Android**: likely `'custom-tabs'` because `chrome.identity` might not open a popup as expected on mobile. We use Chrome Custom Tabs via the mobile browser itself. - For **Firefox on Android**: likely no identity API support, but we can use a `'webview-intercept'` or tab-intercept method (since Firefox for Android extensions can intercept web requests).

The `capabilities` field indicates, for instance, if silent refresh is possible (it is on all, since we manage it), if biometric auth is possible (not in this scope, but maybe in the future we could integrate device biometrics through the browser), secureStorage (Chrome on Android might not allow writing to disk easily for extensions, etc., but MV3 storage should still work).

12.2 Chrome (Android) – Custom Tabs Flow

For Chrome on Android, because an extension cannot open a typical popup window within the Chrome app UI, the approach will be: 1. **Open in Custom Tab**: Launch the OAuth URL using a Chrome Custom Tab. This essentially opens a Chrome-branded in-app browser with the OAuth page, providing a smoother mobile experience than a full external browser jump. 2. **Custom Scheme Redirect**: Use a custom URI scheme or an intent as the redirect target. For example, `saypi://auth/callback` could be a URI we register. The OAuth provider (SayPi's auth server) would be configured to redirect to `saypi://auth/callback` with the code. 3. **Android Intent Handling**: The extension (since it's part of Chrome on Android, which might be tricky because Chrome extensions on Android are limited) – realistically, Chrome on Android currently doesn't support extensions except in some dev environments or Kiwi browser. If it did, catching that scheme might involve the Android system. Alternatively, more practically, we might need the user to complete in the custom tab and come back, then the extension could detect a cookie or so. But for the sake of this PRD, assume we have a way to intercept it: - Possibly using `chrome.tabs.onUpdated` to listen for a navigation in any tab that matches our `saypi://auth/callback` scheme. 4. **Closing the Tab**: Once the code is captured (via the intercept), programmatically close the custom tab if possible. This might require the tab's ID which we can store when opening it. 5. **Continue Flow**: Proceed with token exchange and state update as usual.

The mobile implementation will require thorough testing, as it's inherently less straightforward. If Custom Tabs approach is not feasible due to extension support, we may degrade to simply opening the OAuth page in the default browser and instructing the user to return (which is not ideal, but could be last resort).

12.3 Firefox (Android) – WebRequest Intercept

Firefox on Android does support extensions (through a limited collection), but it might not support the identity API. However, Firefox extensions can use the webRequest API. Our plan for Firefox Android: 1. **Open OAuth in New Tab**: Trigger the OAuth flow by opening a new tab in Firefox with the authorization URL (similar to desktop fallback). 2. **Intercept Redirect**: Use `browser.webRequest.onBeforeRequest` or a similar API to listen for any request to our predefined redirect URI (which in this case might be a web-accessible URI or a custom scheme). 3. **Capture Auth Code**: When the redirect is detected (the request URL contains `?code=` and our known state), extract the code from the URL. 4. **Close Tab**: Use `browser.tabs.remove(tabId)` to close the OAuth tab once we got what we need. 5. **Continue with Token Exchange** as in the normal flow.

This requires that we have the webRequest permission for the redirect domain/scheme. Firefox may allow intercepting requests to certain schemes or to specific domains we control.

12.4 Graceful Degradation and Parity

If any platform does not support the above methods, we will ensure the experience falls back in the least disruptive way: - If the identity API or webRequest intercept is not available, at minimum we ensure that after login the user is instructed to manually return to the app and refresh, **but** we aim to avoid this scenario with the above strategies. - We will document any platform-specific differences. For example, maybe on Chrome Android, the auto-close might not work and the user has to tap a "Return to SayPi" button – if so, that will be clearly communicated in the UI.

Our goal is that by the end of Phase 4 (Mobile Optimization), at least 80% of mobile users can go through the auth flow successfully on the first try, approaching parity with desktop flows.

13. State Management & Broadcast System

To coordinate authentication state across various parts of the extension, we will implement a centralized state management system for auth.

13.1 Centralized Auth State Store

We will create a singleton **AuthStateManager** (in the background script context, or as a module that the background can use and UIs can query). This will hold the current authentication status and user info, and allow components to subscribe to changes.

A simplified TypeScript-like outline:

```
interface AuthState {
  status: 'unknown' | 'unauthenticated' | 'authenticated';
  user?: { id: string; name: string; email: string; /* other profile info */ };
  token?: string; // JWT access token (if needed for reference)
  expiry?: number; // timestamp
}

type AuthStateListener = (state: AuthState, event: AuthEvent) => void;

class AuthStateManager {
  private state: AuthState = { status: 'unknown' };
  private listeners: Set<AuthStateListener> = new Set();

  async initialize() {
    // Rehydrate from storage if available
    const stored = await chrome.storage.session.get('authState');
    this.state = stored.authState || { status: 'unauthenticated' };

    // Set up cross-context sync:
    chrome.storage.onChanged.addListener(this.handleStorageChange);
    chrome.runtime.onMessage.addListener(this.handleMessage);
  }

  getState() {
    return this.state;
  }

  updateState(newState: AuthState) {
    this.state = newState;
    chrome.storage.session.set({ authState: newState });
    // Possibly also update local storage or trigger other side effects
  }

  subscribe(listener: AuthStateListener) {
    this.listeners.add(listener);
  }
  unsubscribe(listener: AuthStateListener) {
    this.listeners.delete(listener);
  }

  private notifyListeners(event: AuthEvent) {
    for (const listener of this.listeners) {
      listener(this.state, event);
    }
  }

  // Example of broadcasting an event to other contexts:
  broadcast(event: AuthEvent) {
    chrome.runtime.sendMessage({ type: 'AUTH_EVENT', event });
    chrome.tabs.query({}, (tabs) => {
      for (const tab of tabs) {
        if (tab.id) {
          chrome.tabs.sendMessage(tab.id, { type: 'AUTH_EVENT', event });
        }
      }
    });
  }

  // Handlers for storage and message events:
  private handleStorageChange(changes, area) {
    if (area === 'session' && changes.authState) {
      this.state = changes.authState.newValue;
      this.notifyListeners({ type: 'STORAGE_SYNC' });
    }
  }
  private handleMessage(message, sender) {
    if (message.type === 'AUTH_EVENT') {
      this.handleAuthEvent(message.event);
    }
  }
  private handleAuthEvent(event: AuthEvent) {
    // Update state based on specific event (LOGIN_SUCCESS, LOGOUT, etc)
    // Then notify listeners.
  }
}
```

The AuthStateManager will be responsible for: - Initializing the state when the extension loads (checking if the user was already logged in from a previous session). - Updating the state when something changes (login success, logout, token refresh). - Storing the minimal state in session storage so that if a new UI context (e.g., the user opens the popup) it can retrieve the current auth state quickly. - Broadcasting events to all contexts (since not all contexts can directly observe changes in session storage, especially content scripts). - Handling external events like storage changes (which might be triggered if, say, the background script updated `chrome.storage.session` and we want that to propagate to any open settings page that might also be reading from storage).

Using both message passing and storage as redundancy ensures robustness: - On Chrome, `chrome.runtime.sendMessage` will wake up contexts that are listening. - As a backup, `chrome.storage.onChanged` can alert any script that perhaps wasn't loaded in time to get the message.

13.2 UI Auto-Update Protocol

All user interface components of the extension will subscribe to auth state changes rather than polling for them or requiring manual refresh. The key UI surfaces include: 1. **Settings Page** – The primary place where sign-in status is shown. It will subscribe to the AuthStateManager (likely via `chrome.runtime.onMessage` for `AUTH_EVENT` messages, or by periodically checking `chrome.storage.session` on focus) to update the displayed user info. 2. **Content Script Overlays** – E.g., the voice input or output overlays in the chatbot pages. These might show a "Sign in to enable voice" message or similar. They should listen for auth events so that if the user signs in via the settings, the overlay can immediately reflect that (perhaps enabling new options or removing a sign-in prompt). 3. **Extension Toolbar Icon** – We could change the icon or badge when the user is signed in (not strictly necessary, but some extensions show a badge number or a different icon state). For example, showing a small "✔" or color change when logged in. If we do this, the background can update the browserAction badge on auth events. 4. **Context Menu Items** – If the extension adds context menu entries that differ when logged in (not currently, but possible), ensure they update accordingly upon auth changes. 5. **Injected Page Elements** – If any part of the web page (via content script) needs to know auth (for example, an embedded "Sign in to use voice here" button), it should also respond to changes.

The goal is a propagation time of under 100ms from the moment auth is completed to when all relevant parts of the extension reflect the new state. This means our message passing and state setting should be done quickly and UIs that receive the event should update immediately (e.g., re-render their React components or update DOM accordingly).

By centralizing state and using a publish/subscribe pattern, we avoid having each component individually query the background or rely on the user's action. It's all event-driven, making the experience feel instantaneous and consistent.

14. Error Recovery & Resilience

No system is perfect, so we need to handle errors gracefully and make the flow resilient to transient problems. We categorize potential errors and define how to handle them:

14.1 Error Classification and Automated Handling

We will classify errors by type and configure an appropriate strategy for each. For example:

```
const errorHandlers = {
  // Transient errors – these can be retried automatically
  'network_timeout':      { retry: true,  maxAttempts: 3, backoff:
'exponential' },
  'service_unavailable':  { retry: true,  maxAttempts: 5, backoff: 'linear' },

  // Authentication errors – require user action
  'invalid_grant':        { retry: false, action:
'reauthenticate' },   // e.g., refresh token expired or code invalid
  'consent_required':     { retry: false, action: 'show_consent' },    // user
needs to grant additional consent or scopes

  // Client-side or configuration errors – log and possibly fallback
  'invalid_request':      { retry: false, action: 'log_and_fallback' },
  'unsupported_browser':  { retry: false, action: 'show_compatibility_notice' }
};
```

In practice: - A **network timeout** or similar (no response from server) will trigger an automatic retry of the last action (like exchanging a token or refreshing). We'll try a few times with increasing delays. If after maxAttempts it still fails, we surface an error to the user ("Network error, please check your connection"). - If the auth server is **down or returning 503**, we might retry a few times but not overload. If persistent, show a user message like "Our servers are busy, please try again in a moment." - For **auth errors** like `invalid_grant` (which often means a refresh token is invalid or the auth code was already used or expired), we cannot solve this by retrying. The user needs to sign in again. In such cases, we will treat it as a logout: clear any partial state and prompt the user to log in fresh (maybe with a message "Your session expired, please sign in again."). - `consent_required` might occur if the user removed a permission or a scope is missing. The action would be to present a prompt informing them they need to grant consent (potentially re-run the OAuth flow with `prompt=consent`). - For **client-side errors** like `invalid_request` (if we sent something wrong) or an **unsupported environment** (user's browser really can't do our flow), we will log these occurrences (for developers to notice) and fall back to a degrade pathway. For example, unsupported browser might show a message: "Your browser does not fully support the sign-in flow. Click here to open the login page manually," as a last resort. - All error messages shown to the user will be friendly and non-technical, while our logs (in telemetry) will record the technical error codes.

14.2 Graceful Degradation

We will build resilience by having fallback behaviors in place: - **No Identity API**: If the `chrome.identity` API is not available or fails, automatically revert to the tab-based flow (Option B). The user might see a full-page login, but we will still attempt to auto-close it and message the extension on completion. - **No PKCE (theoretically)**: If for some reason PKCE isn't supported by the backend (it should be, but if not initially), we will still use state and other mitigations. However, since this is a planned change, PKCE will be required. - **Refresh Failures**: If refreshing a token fails (e.g., device offline when timer triggers), we queue the refresh to try when back online. If a user tries to use the extension during that time, we may attempt an on-demand refresh once connectivity is back. If refresh ultimately fails due to expiration, we catch the `invalid_grant` and prompt login again gracefully. - **Storage Limits**: Extensions have storage quotas. If we ever hit a storage limit error when trying to save tokens (say a very rare case with many tokens or large data), we will clear older data (like any old tokens or redundant info) and try again. The data we store for auth is small, so this is unlikely.

The overarching principle is that any time something goes wrong, we either recover automatically without user noticing, or we present the user with a clear path forward (never a dead end). The extension should not leave the user in a confused state; either they are signed in, signed out, or explicitly told that an error occurred and how to address it.

15. Analytics & Experimentation

To measure success and continually improve, we will instrument the new authentication flow with analytics events and possibly run A/B tests on certain variables.

15.1 Key Metrics to Track

We will collect (in accordance with privacy policies) anonymous, aggregated metrics around the auth funnel and usage. Some key metrics and their definitions:

```
interface AuthMetrics {
  // Funnel metrics
  promptShown: number;    // how many sign-in prompts (of any kind) were shown
  promptClicked: number;  // how many prompts resulted in the user clicking
"Sign In"
  authStarted:
number;    // how many sign-in flows were initiated (user clicked Sign In)
  authCompleted:
number;  // how many sign-in flows completed successfully (user got a token)

  // Performance metrics
  timeToAuth: number[];   // collection of durations from click "Sign In" to
auth complete (for percentile calc)
  refreshSuccess: number; // count of successful token refresh events
  refreshFailure:
number; // count of failed refresh attempts that required user re-login

  // Engagement metrics
  featuresUsedPostAuth: string[]; // which premium features users engage with
after signing in (to gauge what drove sign-in)
  returnUserAuth: number;        // count of returning users who remain
authenticated (didn't have to log in again)

  // Error metrics
  errorsByType: Record<string, number>; // counts of occurrences of different
error types (network, invalid_grant, etc.)
  recoverySuccess: number;  // how many errors were recovered automatically vs
requiring user action
}
```

From these metrics we can derive: - **Conversion rates**: e.g., promptShown vs authCompleted gives an overall conversion of prompts to successful sign-ins. We want to see this high, and specifically for engaged users (5+ interactions) we targeted 70%+ sign-in. - **Drop-off points**: If `promptClicked` is high but `authStarted` is low, maybe users click sign-in but then abandon (meaning the flow might be failing to open or is too slow). - **Time to Auth**: We'll look at P50, P90, P99 times. For instance, a goal might be P90 < 15 seconds. If we see many outliers (very long times), we investigate what causes the delay (likely network or user taking time). - **Refresh stats**: Ideally, refreshSuccess >> refreshFailure. If we see refreshFailure > 0.5% of total attempts, that might indicate issues. - **Feature adoption**: Knowing which features are used after auth helps validate our value proposition. E.g., if "enhanced_voice" is heavily used post-auth, that's likely a driver. - **ReturnUserAuth**: If many return users are not authenticated (meaning our silent refresh didn't keep them logged in), that's a problem to fix. We set a goal (<5% require re-auth in 30 days, meaning 95% stay logged in for at least a month unless they log out).

These metrics will be sent to our analytics backend with appropriate anonymization (no personal data, just event counts and maybe hashed IDs to correlate a user's funnel anonymously).

15.2 A/B Testing Framework

We will utilize our experimentation framework to test variants of the authentication prompts and flows to optimize conversion. Potential experiments:

```
const experiments = {
  'auth_trigger_timing': {
    control: 5,       // show first prompt after 5 interactions
    variant_a: 3,     // variant: show after 3 interactions (earlier)
    variant_b: 10     // variant: show after 10 interactions (later)
  },
  'cta_text': {
    control: 'Sign In',
    variant_a: 'Unlock Premium Features',   // different call-to-action text on
button
    variant_b: 'Save Your Settings'
  },
  'auth_providers_order': {
    control: ['google', 'github', 'email'],             // order in the login
UI
    variant_a: ['email', 'google', 'github']            // maybe putting email
first vs SSO first
  }
};
```

We will randomly assign users (or sessions) to different variants and track conversion metrics. For example: - If showing the prompt earlier (3 interactions) leads to a higher overall sign-in rate or not. We have to balance prompting too early (might annoy) vs too late (might lose users who drop off). - The text of the sign-in call to action might influence whether users click it. “Unlock Premium Features” might entice more than just “Sign In”. - The order of provider options might affect conversion (some users might go with the first option presented).

The coding implementation means: - The extension will have some means (config flags or a simple experiment config fetched from storage) to decide which variant to use for a given user. - It will then use the thresholds or text accordingly. - We’ll gather metrics segmented by variant to compare.

The primary goal of experimentation here is to optimize the prompt strategy for maximum sign-in conversion without harming user experience.

16. Implementation Phases

Delivering this project will be done in iterative phases, each with its own focus and success criteria. This allows testing and validating each component before the next.

Phase 1: Core Identity API Integration (Week 1–2)

- **Tasks**: Implement the basic OAuth flow via `chrome.identity.launchWebAuthFlow` with PKCE on desktop. Set up the backend token exchange endpoint (in coordination with backend team) if not already available. Handle the response and store tokens. Also implement the fallback tab flow (Option B) as a backup path.
- **Success Criteria**: By end of Phase 1, 90%+ of desktop users (in a test group or internal testing) can complete the authentication flow successfully using the new method. The settings UI should update automatically on login for desktop Chrome and Firefox.

Phase 2: State Broadcasting System (Week 2–3)

- **Tasks**: Build out the AuthStateManager in the background. Implement message broadcasting and storage syncing for auth state. Update the settings page UI code to subscribe to auth updates and re-render accordingly. Create visual indicators (spinner, etc.) for in-progress and success states.
- **Success Criteria**: Auth state changes propagate to UI within <100ms. In testing, when we simulate a token acquisition or expiration, the settings page (and any other test surfaces) update almost instantly. No manual refresh needed in any tested scenario.

Phase 3: Progressive Authentication Prompts (Week 3–4)

- **Tasks**: Implement engagement tracking (counters for usage). Develop the toast notification component and modal dialog for prompts. Integrate logic to trigger these at the defined thresholds (5 uses, 10 uses, premium attempt). Ensure that clicking “Sign In” on these prompts invokes the Phase 1 auth flow. This phase also involves some UX design work (copy for prompts, styling).
- **Success Criteria**: In a controlled test, 70% of users who reach the prompt (meaning they used the extension enough to be prompted) proceed to sign in. This will be measured in a small beta. The prompts should appear at correct times and be dismissible as designed, with no major UX bugs.

Phase 4: Mobile Optimization (Week 4–5)

- **Tasks**: Implement platform detection and alternate flows for Chrome Android and Firefox Android as outlined. Test on actual devices or emulators for Android Chrome (using a compatible browser like Kiwi if needed) and Firefox. Ensure tabs are closing and codes are captured. Make any necessary adjustments to the fallback flow to make it mobile-friendly (e.g., perhaps using a different approach to close tabs on mobile).
- **Success Criteria**: At least 80% of sign-in attempts on supported mobile platforms succeed on the first try. Specifically, test that a user can sign in on Firefox for Android and Chrome for Android (or compatible) without getting stuck. The experience might not be as slick as desktop, but it should be functional and not require exotic user steps.

Phase 5: Silent Refresh & Token Management (Week 5–6)

- **Tasks**: Implement the silent token refresh logic using alarms. Test that refresh happens on schedule. Implement refresh token rotation logic (this likely involves coordination with backend—ensuring the token endpoint issues a new refresh token each time). Also implement handling for refresh failures (with a simulated expired token). Ensure that if the user manually logs out or if refresh fails, appropriate cleanup is done.

- **Success Criteria**: Less than 1% of sessions end up with an unexpected expiration. In testing, if we artificially shorten token life, the extension successfully refreshes tokens behind the scenes the majority of times. Users in the beta program report that they didn't have to frequently log back in – the session "just stays logged in" day to day. Also ensure no memory leaks or excessive alarms (like if many are set, they cancel properly after logout).

Each phase will be accompanied by internal testing and possibly a small percentage beta rollout to gather data. We will only proceed to a wider launch once these phase gates are successful.

(Note: The timeline is an estimate; actual scheduling might adjust, and some phases could be done in parallel by different team members. But this breakdown provides a guiding structure.)

17. Success Criteria & Key Metrics

To know when this project is truly successful (beyond just meeting the functional requirements), we define some Key Performance Indicators (KPIs) and targets:

Primary KPIs

- **Sign-In Conversion Rate**: Increase the proportion of engaged users (for example, users with >5 interactions) who eventually sign in from ~30% to **70% or more**. This will be measured via analytics over a period after release.
- **Auth Completion Rate**: >85% of users who initiate the sign-in process (click "Sign In") successfully complete it. (This accounts for drop-offs; some may still cancel, but we want the flow itself to not be the cause of drop-off beyond a small percentage).
- **Time to Auth**: The 90th percentile time from when a user clicks "Sign In" to when the extension shows them as authenticated should be **15 seconds or less**. (This includes the user actually completing their login—since external factors like user reading time and entering credentials vary, P90 of 15s is a reasonable target indicating the flow is efficient. P50 might be ~5s).
- **Silent Refresh Success Rate**: >99% of token refresh operations occur without requiring the user to log in again. In other words, less than 1% of sessions should expire and force the user to reauthenticate unexpectedly.

Secondary KPIs

- **Return User Continuity**: <5% of returning users (within a 30-day window) find themselves logged out and have to sign in again. Ideally, once signed in, users remain so for at least 30 days unless they manually log out.
- **Error Recovery Rate**: >90% of transient errors (network issues, etc.) are resolved automatically by the system (retries or fallbacks) without the user seeing a failure. Our goal is that users rarely encounter an error message; the system handles hiccups.
- **Mobile/Desktop Parity**: The mobile auth success rate should be within 10% of desktop's rate. For example, if 90% of desktop users succeed on first try, we want at least ~80% on mobile. We don't want mobile users left behind due to technical issues.
- **Feature Adoption Post-Auth**: >60% of users who sign in go on to use a premium feature (e.g., try an enhanced voice or exceed the free quota). This indicates that the sign-in is actually unlocking

value and that we prompted them at the right time (they had a need). If this number is low, it might mean users sign in but still don't use premium features, questioning the value proposition timing.

These metrics will be monitored through analytics dashboards post-release. We will consider the project "done" from a product perspective when the primary KPIs are met or exceeded without regressing user experience.

18. Testing Strategy

A comprehensive testing plan is needed due to the many moving parts (OAuth flow, storage, multi-surface UI, etc.). We will employ multiple levels of testing:

18.1 Unit Testing

- **AuthStateManager**: Unit test the state management logic – subscribing, storage sync, event handling. Simulate changes and ensure listeners get updates.
- **Token Utilities**: Test functions for generating PKCE challenges/verifiers, state values, and any crypto utilities to ensure they meet requirements (correct length, character set, etc.).
- **Error Handler**: Test the error handling logic by simulating different error codes and verifying the correct action is taken (e.g., a `network_timeout` triggers retries with backoff, an `invalid_grant` triggers a reauth prompt, etc.).
- **Analytics Utils**: If there are functions that throttle or batch analytics events, test those to ensure metrics are recorded correctly.

18.2 Integration Testing

We will use automated integration tests (possibly with Selenium or Puppeteer on a test browser, or Firefox's extension test framework) to simulate real flows: - **Full Auth Flow (Desktop)**: Simulate a user clicking sign-in, going through an OAuth login (this might be mocked or a test OAuth endpoint), and ensure the extension receives the token and updates the UI. We can automate a Google OAuth flow using a test Google account if feasible, or use a stub identity provider. - **Token Refresh Cycle**: Issue a short-lived token (e.g., expire in 2 minutes) in a test environment, then let the extension run and ensure that it calls the refresh endpoint and obtains a new token without user action. Verify the user remained "authenticated" throughout. - **State Broadcast**: Start with extension in logged-out state, then programmatically trigger a login (or simulate receiving a token) and verify that both the settings page and a content script (simulated) receive the update. Similarly, simulate logout events. - **Mobile Flow Simulation**: This is harder to automate. We might use a headless Android emulator or a mobile-configured Selenium to test the fallback. At minimum, test the custom scheme intercept logic (maybe in a desktop environment by forcing the fallback path). - **Multiple Provider Flow**: If supporting Google, GitHub, and email login, ensure each can complete (this might involve stubbing or using test accounts).

18.3 End-to-End (E2E) Testing

Manual and beta testing with real flows: - **New User Journey**: A tester (or beta user) installs the extension fresh, uses it unauthenticated for a while (triggering the progressive prompts), then signs in when prompted, then uses a premium feature. Verify everything works in sequence and the user experience is smooth. - **Returning User with Expired Token**: Sign in, then manually expire the token (maybe via an admin tool or by waiting if the token is short-lived) and then see what happens when the user tries to use the extension. Ensure the silent refresh either kept them signed in or they are prompted nicely to sign in again. - **Premium Feature Gating**: Try to use a premium voice or exceed the free quota without signing in. Confirm that the extension blocks the action and shows the appropriate modal prompt. Then sign in via that prompt and confirm the feature immediately works after. - **Cross-Device Scenario**: Sign in on Chrome desktop, then install the extension on Firefox (or Chrome mobile) for the same account – ensure that the user can sign in on the second device and that usage counts (for prompts) are separate per device (since not easily shared without a backend). - **Logout and Re-login**: Test that logging out from the extension truly clears everything (the user should have to enter credentials again if they log back in) and that the UI immediately shows logged-out state.

Additionally, security testing (by the security team) will be done on the OAuth implementation and token storage to ensure no leakage.

19. Rollout Strategy

Given the impact on the user experience, we will roll this out in stages to mitigate risk:

Staged Rollout Plan

1. **Internal Alpha (5% of users)**: Initially, release the new flow to internal team members and a small set of friendly users (via a feature flag or an "early access" toggle in the extension). This allows testing in real conditions and gathering feedback/metrics. We especially want to monitor for any login failures or edge-case issues here.
2. **Public Beta (20% of users)**: Gradually enable for more users, possibly region-wise or randomly. At this stage, have in-app feedback channels or closely watch support tickets. Ensure that our success metrics are trending in the right direction (or at least not worse than before) before going wider.
3. **General Availability (100%)**: Once confident, flip the switch for everyone in a new extension version update. Communicate changes in release notes (highlighting the improved sign-in flow and any new features like being able to use the extension a bit without signing in).

During the rollout, we will keep the old mechanism available as a fallback (for example, the extension might still accept the old cookie method if the new flow fails, just to not break anyone). But our aim is to not need it.

Rollback Criteria

We define clear criteria to decide if we need to pause or rollback the rollout: - If the **authentication success rate** (users who attempt login and succeed) drops below 60% at any point (meaning our flow might be failing a large fraction of users), we will halt and investigate. - If the **error rate** (any critical error that prevents login) exceeds 5% of attempts, that's a red flag. - If we see performance issues like P99 time to auth spiking above 30 seconds, that indicates possible hangs or timeouts affecting users – another reason to pause. - Significant new support volume or user complaints about not being able to log in would also prompt a rollback.

Rollback would involve either switching all users back to the old flow (if feasible via config) or in worst case, releasing a new version that reverts to the old flow, until fixes are made.

The rollout and monitoring will be closely coordinated with the support team and possibly with some end-user communication (we might post an announcement or in-app notice, especially about the progressive free-tier usage change so users aren't caught off guard by the new prompts).

20. Security & Privacy Considerations

Security is a first-class concern in this implementation, as it deals with authentication tokens and user accounts.

Required Security Measures

- **HTTPS Only**: All network calls for authentication (token exchange, refresh, etc.) must use HTTPS endpoints. We will enforce this in code (no calls to non-HTTPS).
- **PKCE Everywhere**: As stated, PKCE is required for all OAuth flows. We will not allow falling back to implicit grants or other less secure flows.
- **State Parameter Mandatory**: Every OAuth request must include a cryptographically strong `state` value, and the response must be validated against it. No exceptions.
- **Token Handling**: Access tokens or refresh tokens should never be logged (even in debug logs). We will be careful not to include them in any error reporting or analytics. If an error occurs that involves a token, we'll sanitize it.
- **Encrypted Storage**: Refresh tokens (which are long-lived and sensitive) will be stored encrypted if placed in persistent storage. Access tokens are short-lived and will be in session memory, which is lost on restart (improving security).
- **Token Revocation**: Implement the ability to revoke tokens (if backend provides an endpoint or if not, at least clear them). In cases of suspicious activity (e.g., if we detect a refresh token was used from another location, indicating possible theft via token reuse detection), we will treat it as a potential breach: log the user out of all devices and require re-auth.
- **Content Security Policy**: Ensure the extension's CSP is updated to allow the needed OAuth domains for `launchWebAuthFlow` (these calls might not require CSP changes since it's out-of-context, but any web accessible resources we use should be whitelisted properly and safely).
- **No Passwords**: The extension itself will never handle user passwords directly (the OAuth web page does that). This is good for security – we just need to make sure we maintain that separation.

Privacy Considerations

- **Minimal PII**: The extension should not collect more personal data than necessary. It really only needs to know an identifier for the user (like user ID or email) after login, and perhaps their subscription status. We will not expose or use personal details beyond what's needed for the UI (e.g., greeting the user by first name).
- **Data Retention**: We will clarify how long we keep authentication data. For instance, tokens are kept until they expire or user logs out. Any analytics events are aggregated and not tied to personal identities.
- **Consent**: Since we are encouraging sign-in more actively, we should ensure the user is informed about what signing in means in terms of data. We will review the onboarding text and possibly add a link to a privacy policy or a note like "By signing in, you agree to...". Because GDPR/etc., if we're storing their usage data tied to an account, we need consent. The sign-in inherently covers some of this, but we should be transparent.

- **Account Deletion**: Ensure that if a user deletes their account (through the website, presumably), and they continue using the extension, the extension handles that gracefully (they might get logged out). Also, provide info in user documentation on how to disconnect or delete data (likely by deleting account on site).
- **Telemetry Opt-Out**: Respect any user settings about analytics. If the user has opted out of analytics/telemetry, we should disable the tracking of metrics for that user's extension (except essential operational logs).

In summary, the new auth flow should greatly improve UX but we must ensure it doesn't compromise on security and respects user privacy. We'll have a security review done before launch to double-check all these points.

21. Documentation Requirements

To support both internal developers and end users, we will prepare documentation alongside implementation:

Developer Documentation (for internal use and open-source, if applicable)

- **Auth Flow Integration Guide**: Document how the extension integrates with each auth provider. For instance, detail the OAuth endpoints, parameters used, and how to add a new provider if needed in the future.
- **State Management Architecture**: Explain how the AuthStateManager works, with diagrams possibly showing message flow between background and content scripts. This helps future developers understand the event-driven model we used.
- **Error Handling Flowchart**: Provide a simple flowchart or table mapping error codes to actions (similar to the errorHandlers table) so that it's clear how different errors are meant to be handled.
- **Platform-Specific Notes**: Write up the special cases for Chrome Android and Firefox Android (and any other quirks discovered during development). This should include how the custom tab or webRequest intercept works, and any limitations (e.g., "on Firefox Android, window.close might not close the tab automatically; user might have to close it" – if that's the case).
- **API Contract**: Document the backend API endpoints used (`/oauth/authorize`, `/oauth/token`, logout/revoke if any) so that backend and future devs have a reference of what the extension expects.

User Documentation (for external users, likely on the website or help center)

- **"Why Sign In?" FAQ or Help Article**: Explain the benefits of signing in to the extension (e.g., saved settings, access to premium features, synchronization, etc.) to reinforce the prompts in a place users can read more.
- **Privacy & Security FAQ**: Reassure users what data is stored, how their credentials are handled (or not handled by us), etc. Given we highlight security messaging in the UI ("We never see your password"), we should have an article that expands on that trust statement.
- **Troubleshooting Guide**: A help document for if the user cannot sign in. This should list common issues (pop-up blocked, company Google account not allowed, etc.) and what to do. Also instructions like "If you sign in but the extension still says not signed in, try refreshing the settings page" (hopefully that never happens with the new flow, but just in case).

- **Account Management Instructions**: Since users can now more easily sign in, some might ask, “How do I sign out?” or “How do I change accounts?”. The extension UI will have sign out, but document that. Also if they want to fully delete their SayPi account, instruct them to do that on the website.

Having these documentation pieces ready by launch will help ensure smooth adoption and reduce confusion. Support team should be trained with the new flow details as well.

Appendix A: Technical Specifications

OAuth 2.1 Compliance Checklist

- **PKCE Mandatory** – The extension only uses Authorization Code flow with PKCE, in line with OAuth 2.1 recommendations (no implicit flow).
- **Refresh Token Rotation** – Implemented to prevent reuse of refresh tokens.
- **No Implicit Grant Used** – We have completely avoided the deprecated implicit token flow.
- **Redirect URI Exact Matching** – We use a fixed redirect URI via `getRedirectURL()` or custom scheme, which is registered and exact (no wildcards that could be exploited).
- **State Parameter Required** – Every auth request includes a state and we verify it on return.

Browser API Compatibility Matrix

Feature	Chrome Desktop	Chrome Android	Firefox Desktop	Firefox Android
`identity.launchWebAuthFlow`	Supported	Not supported	Supported	Not supported
Custom Tabs (for OAuth)	N/A (desktop)	Used	N/A	⚠ Partial (Firefox might not support custom tabs, using alt method)
`chrome.alarms` (background)	Yes	Yes	Yes	Yes
`chrome.storage.session`	Yes	Yes	(as `browser.storage`)	Yes
`webRequest.onBeforeRequest`	Yes	N/A (not needed)	Yes	Yes (for intercept)

Notes: Chrome Android currently doesn’t officially support extensions; in contexts where it does (like Kiwi browser or future support), our approach uses Custom Tabs. Firefox Android allows extensions but with limitations; we rely on webRequest API there.

Appendix B: Standard Error Codes & Messages

For consistency, the extension will use a set of standard user-facing error messages for common auth failures:

Error Code	User Message	Recovery Action
AUTH_001	"Connection timeout. Retrying…"	Auto-retry with exponential backoff (user sees spinner/ message)
AUTH_002	"Session expired, please sign in again."	Show sign-in prompt (user needs to reauthenticate)
AUTH_003	"Browser not supported for sign-in."	Show a notice explaining the issue and possible alternatives (perhaps direct them to use a supported browser)
AUTH_004	"Premium feature – Sign in to continue."	Show the value proposition modal and require sign-in to proceed

These codes/messages will appear in logs or UI as appropriate. Having them standardized ensures users get a clear, friendly message and we can internationalize them if needed in the future.

This document provides a consolidated specification for the improved authentication flow of the SayPi extension. It integrates the original requirements with new enhancements focused on progressive engagement and robust OAuth implementation. Developers should use this as the guiding reference for implementation, and testers should validate against the acceptance criteria and KPIs outlined.

---

1 auth-signin-flow-prd.md
file://file-XE23rbXNwHvs4cuRtLP5r3