Micro-Frontends vs. Iframes for App Integration: A Quick Evaluation

What Are Micro-Frontends?

Micro-frontends are an architectural approach where a web app is split into smaller, independently deployable frontend applications that together form a single UI (Micro Frontends). In practice, this means different teams or modules can build and deploy their part of the UI (e.g. one in React, another in Angular) without a tightly coupled monolithic codebase. The key idea is that these micro-apps are composed in the browser to appear as one seamless application to users. This approach can improve modularity and enable separate development streams, much like microservices do on the backend.

Communication in Micro-Frontend Architectures

Because each micro-frontend is its own application, a common question is how they communicate or share data. In general, micro-frontends **minimize direct communication** to avoid tight coupling (Micro Frontends). When communication is needed, typical strategies include:

- **Custom Events:** Micro-apps can publish and listen for DOM events (or a pub-sub system) as an indirect communication channel (Micro Frontends). For example, a "userLoggedIn" event can be dispatched by a login micro-frontend and caught by others. This keeps modules loosely coupled, though it requires clear conventions on event names and payloads.
- **Props/Callbacks via a Container:** If using a micro-frontend framework or library, the shell (container app) can initialize micro-apps with certain data or callbacks. This is akin to React props – e.g. passing a function from the container to a micro-app to call when it needs something. It makes the interface between parts more explicit.
- **Global State or Shared Store:** In some cases, a global state management (like a Redux store) is shared. However, this is used carefully since it introduces coupling; often each micro-frontend keeps its own state store to remain self-contained (Micro Frontends). Shared state is only used for truly global concerns (e.g. user session info), and even then events or APIs are preferred over directly mutating a common store.

In summary, micro-frontends tend to communicate via browser APIs or through the framework's mechanisms rather than direct function calls across boundaries, keeping each piece as isolated as possible.

Why Iframes Cause Issues (CORS, OAuth2, X-Frame-Options)

Integrating a second app via an `<iframe>` provides strong isolation but also brings specific challenges. By design, an iframe keeps the embedded app separate from the parent page, which leads to several integration hurdles:

- **Cross-Origin Resource Sharing (CORS):** If the main app and the iframe content are on different domains (or ports), the browser's same-origin policy restricts interactions. Any API calls from the iframe to the main app's domain (or vice versa) are treated as cross-origin requests and must be permitted by CORS headers. Sharing data or calling APIs across the boundary requires careful setup (or hacks like `postMessage`). One developer notes that with iframes "things like CORS and communication [between frames]" become hard to manage (html - Micro frontend architecture advice - Stack Overflow). In contrast, micro-frontend architectures typically load all micro-apps into the same page context, often under the same origin, which **reduces or avoids CORS issues**. For example, using a build tool like Module Federation can fetch a micro-app's code from another origin but execute it in the host page, avoiding cross-domain scripting at runtime (Micro Frontend Architecture (Iframes x Module Federation) | by Vinicius Marson | Medium). Additionally, many teams use a reverse proxy so that each micro-frontend is served under a unified domain path (e.g. `mainapp.com/app1` and `mainapp.com/app2`), eliminating cross-origin calls from the browser's perspective (How do you share authentication in micro-frontends - DEV Community). (You may still need to configure CORS on static asset requests, but this is easier than handling persistent cross-window communication in iframes.)

- **OAuth2 Authentication Flows:** OAuth2 (and OpenID Connect) often involves redirects to an identity provider (IdP) or third-party login page. Many IdPs set the header **X-Frame-Options** to `DENY` or `SAMEORIGIN` on their login pages, which prevents them from being displayed inside an iframe on a different site (X-Frame-Options - HTTP | MDN) (javascript - Refused to display...in a frame because it set 'X-Frame-Options' to 'same origin' in React & oAuth - Stack Overflow). This means an OAuth login popup or redirect will simply not show or will be blocked if your second app is running inside an iframe. A Stack Overflow discussion concluded that there is "no good solution" to fully handle OAuth flows inside a locked-down iframe, aside from breaking out of the frame (opening a new window for login) (javascript - Refused to display...in a frame because it set 'X-Frame-Options' to 'same origin' in React & oAuth - Stack Overflow). With micro-frontends, this problem is largely avoided. Since micro-frontends are part of the host page, they can trigger OAuth flows in the top window as a normal redirect or popup, which IdPs allow. All micro-frontends can share the authenticated session of the user (e.g. via a shared cookie or token storage on the same domain) once the user logs in. In short, a micro-frontend architecture enables a **single sign-on** experience: the user logs in once, and all parts of the application (each micro-app) recognize the session. There's no need to embed a login flow in a hidden iframe because the micro-apps run as first-party components of the site.

- **X-Frame-Options Restrictions:** Aside from OAuth providers, your own integrated app might send X-Frame-Options headers (for security against clickjacking) that prevent it from being framed in another application. If the second app was not originally intended to be embedded, it might default to `SAMEORIGIN` policy, causing the exact "refused to display in a frame" errors. Micro-frontends, by doing away with iframes, inherently bypass any X-Frame-Options issues. The content is not being embedded as an external page; it's being assembled as part of the single-page application. This means you don't have to relax security headers to integrate the apps. The micro-frontend approach **sidesteps frame restrictions** by design – there is no iframe container, so X-Frame-Options never comes into play for your internal micro-apps.

How Micro-Frontends Mitigate These Issues

Adopting micro-frontends can solve or reduce the above problems through a combination of architectural choices and tools:

- **Single DOM/Window Context:** All micro-frontend parts run in the same page context, so they can interact like normal JavaScript modules. This avoids the sandbox of an iframe and its attendant cross-window communication headaches. For instance, sharing a user token from one micro-app to another can be as simple as calling a function or using a mutual storage (provided you've designed a way to do so), rather than messaging between frame boundaries. As a result, issues with CORS and third-party cookies are minimized because everything is happening in a first-party context (especially if served under one domain).

- **Module Federation and Similar Tools:** Modern micro-frontend implementations often use tools like **Webpack Module Federation** (or analogues in other build systems) to load remote micro-apps at runtime. This allows the micro-app's code to execute inside the main application's JavaScript context. The benefit is a **deeper integration**: it "avoids iframe-specific challenges like ... cross-origin communication" by letting the micro-frontend code call the same APIs and use the same global objects as the host app (Micro Frontend Architecture (Iframes x Module Federation) | by Vinicius Marson | Medium). In practice, the host app might pull in a remote module (say, the entire second app's bundle) with CORS-enabled script fetch, then bootstrap it as part of the page. From that point on, that micro-app behaves as if its code were part of the host application, eliminating frame barriers.

- **Custom Events and Global Pub-Sub:** As mentioned, micro-frontends commonly use custom browser events to communicate (Micro Frontends). This is a straightforward way to have, for example, the main app notify all micro-apps of a theme change or user action. Unlike with iframes, you don't need

`postMessage` to cross a window boundary – events and JavaScript objects are all in one page. This drastically simplifies communication logic (and removes the need for CORS entirely in most intra-app messaging, since no HTTP requests are involved in local messaging).

- **Unified Routing and Proxying:** A common strategy is to use a **reverse proxy or path-based routing** on the web server or CDN level to serve all micro-frontends under a single origin (How do you share authentication in micro-frontends - DEV Community). For example, the main app could live at `example.com`, and the second app could be deployed at `example.com/second-app` (even if internally it's hosted elsewhere). To the browser, everything is coming from `example.com`, so embedding and resource sharing are seamless. This setup means you won't hit CORS or X-Frame-Options issues because from the client perspective there's no cross-origin at all. Each micro-frontend can still be developed and deployed independently behind the scenes, but the user gets one unified domain.

- **Shared Authentication (SSO):** In micro-frontend architectures, authentication is often centralized. You might have a dedicated auth micro-frontend or a shared auth service, but crucially, once a user logs in, their session/token is stored in a way all micro-apps can access (for example, an HTTP-only cookie on the common domain or a token in `localStorage`). This means subsequent micro-frontends won't need a separate OAuth flow; they'll detect the existing login. If using an OAuth2/OIDC provider, you can perform the login redirect in the main app (or a specific micro-frontend responsible for login) and then distribute the credentials. Because there is no iframe involved, the login page won't be blocked. In scenarios where different microfrontends are on different subdomains, a common technique is to use a **parent domain cookie** for the OAuth session (e.g. set cookie on `.example.com` so that `app1.example.com` and `app2.example.com` both see it). This avoids the third-party cookie problems that iframes would encounter and ensures a one-time sign-in for the user across the micro-apps.

- **Web Components or JavaScript APIs:** Some micro-frontend frameworks leverage Web Components or similar standards to encapsulate micro-apps. For instance, a micro-frontend could be delivered as a custom HTML element (`<my-micro-app></my-micro-app>`). These still run in the main DOM and thus bypass iframe restrictions, while providing style and script isolation. Communication can be done via custom element attributes, DOM events, or a shared global. The key point is that these standards-based approaches still keep everything on the client side in one origin, thereby reducing CORS and X-Frame issues.

Conclusion: Are Micro-Frontends a Suitable Solution?

Micro-frontends are likely to solve the specific integration issues you're encountering with iframes. They provide a way to integrate a second app without the hard separation that an iframe imposes, thereby avoiding the **X-Frame-Options login blocker** and making CORS management more straightforward (often unnecessary) at the UI level. By assembling the apps into a single frontend application, you can handle OAuth2 flows in a unified way and share authentication state across modules more easily. In short, the problems with "app-in-an-iframe" integration – cross-origin script limits, clunky communication channels, and framing restrictions – are largely eliminated when you switch to a micro-frontend architecture.

However, it's worth noting that micro-frontends come with their own complexity and overhead. Adopting this approach is a significant architectural decision; it introduces considerations around build processes, deployment coordination, and performance (loading multiple micro-apps). In scenarios with only a couple of frontend modules and a single team, some engineers caution that a full micro-frontend architecture might be overkill (html - Micro frontend architecture advice - Stack Overflow) (Are microfrontend a viable architecture for real world apps? : r/react). That said, given your experience with React and Angular, you would be leveraging known tools (like module federation or single-spa) to implement it. With proper planning, those tools can **overcome the integration pains** you described and allow the two apps to coexist as one product without the iframe drawbacks.

Bottom line: Micro-frontends can be a suitable solution to your CORS, OAuth2, and X-Frame-Options problems. They offer a more elegant integration path than iframes by avoiding cross-origin iframes entirely. If you need tight integration and a smoother user experience between your main app and the new module, micro-frontends are worth considering. Just weigh the added complexity – if those issues are critical blockers right now, the architectural shift can be justified. Overall, you can expect significantly fewer cross-domain headaches with a micro-frontend approach than with an iframe-based embedding (html - Micro frontend architecture advice - Stack Overflow) (Micro Frontend Architecture (Iframes x Module Federation) | by Vinicius Marson | Medium), so it is likely the right direction to resolve the challenges you're facing.

Sources:

- Fowler, M. *Micro Frontends* – Definition of micro-frontend architecture (Micro Frontends)
- Stack Overflow – Discussion of iframe drawbacks (CORS, communication, cookie sharing) (html - Micro frontend architecture advice - Stack Overflow) (html - Micro frontend architecture advice - Stack Overflow)
- Tyagi, U. – *Module Federation vs. Iframes* (Medium) – Notes that module federation avoids iframe cross-origin issues (Micro Frontend Architecture (Iframes x Module Federation) | by Vinicius Marson | Medium)
- Stack Overflow – *OAuth in iframe issue* – OAuth2 login blocked by X-Frame-Options, no good workaround inside iframe (javascript - Refused to display...in a frame because it set 'X-Frame-Options' to 'same origin' in React & oAuth - Stack Overflow) (javascript - Refused to display...in a frame because it set 'X-Frame-Options' to 'same origin' in React & oAuth - Stack Overflow)
- Klee, E. – *Sharing Authentication in Micro-Frontends* (Dev.to) – On using single domain (reverse proxy) to avoid cross-origin in micro-frontends (How do you share authentication in micro-frontends - DEV Community)
- Fowler, M. – *Micro Frontends* – On micro-frontend communication via events and avoiding tight coupling There isn't a "one-true" way to build micro-frontends —they're an architectural pattern that can be implemented using different techniques depending on your needs. Here's a high-level comparison of the main approaches:

---

1. Iframe-Based Integration

How It Works:
Each micro-frontend runs in its own isolated iframe. The container (shell) loads the appropriate iframe based on routing.

Pros:
• **Strong Isolation:** CSS, JavaScript, and DOM are completely separated, reducing conflicts.
• **Legacy Support:** Can integrate older or very different applications without much rework.

Cons:
• **Performance Overhead:** Each iframe has its own browser context, leading to duplicated resource loads.
• **Communication Challenges:** Sharing data (e.g., authentication, state) between the shell and the iframe requires extra messaging (like postMessage), which can be cumbersome.
• **Limited Integration:** Seamless user experience (like shared navigation or animations) can be harder to achieve.

---

2. Client-Side Composition

This is the most common modern approach. The container application loads micro-frontends dynamically into a single-page application.

There are several variations:

a. Using JavaScript Orchestration

• **Method:** The container loads micro-frontend bundles (via `<script>` tags) and calls global functions (e.g., `window.renderX`) to mount each part into designated DOM nodes.
• **Pros:**
  – Easy to develop and test micro-frontends independently (each can also run standalone).
  – More control over when and how components are mounted.
• **Cons:**
  – The container must know the entry-point contracts, which requires coordination.

b. Using Web Components

• **Method:** Each micro-frontend is encapsulated as a custom element (using the Web Components spec). The container simply instantiates these elements.
• **Pros:**
  – Native encapsulation (Shadow DOM) helps prevent CSS and script conflicts.
  – The shell doesn't need to know about the internal implementation details.
• **Cons:**
  – Browser support for advanced features (or polyfills) may be required.
  – Communication between components is still needed for cross-cutting concerns.

c. Using Frameworks and Libraries (e.g., Single-SPA, Qiankun)

• **Method:** Dedicated orchestration frameworks help register, mount, and unmount micro-frontends.
• **Pros:**
  – Provides standardized routing, lifecycle management, and communication patterns.
  – Allows teams to work with different frameworks within the same container.
• **Cons:**
  – There is a learning curve and potential configuration overhead.
  – Debugging cross-framework issues can be challenging.

d. Using Module Federation (Webpack 5)

• **Method:** Module Federation enables dynamically loading remote modules at runtime while sharing common libraries (like React).
• **Pros:**
  – Reduces duplicate dependencies by sharing libraries across micro-frontends.
  – Supports independent deployments with runtime integration.
• **Cons:**
  – Requires careful version management and bundler configuration.
  – Tightly couples the container's expectations with the exposed API of the micro-frontend.

---

3. Server-Side Composition

How It Works:
The server aggregates HTML fragments or renders the complete page by calling individual micro-frontend services before sending it to the browser.

Pros:
• **SEO & Initial Load:** Pre-rendered content can be faster to display and better for SEO.
• **Consistent Experience:** Provides a cohesive experience since everything is stitched together before reaching the client.

Cons:
• **Increased Complexity:** Requires orchestration logic on the server and often tight coupling with server infrastructure.
• **Deployment Overhead:** Independent deployments can be more challenging as the server must coordinate versioning and integration.

---

4. Edge-Side Composition

How It Works:
Similar to server-side composition, but the aggregation happens at the CDN or edge (using technologies like AWS Lambda@Edge or Cloudflare Workers).

Pros:
• **Performance:** Can reduce latency by composing pages closer to the user.
• **Scalability:** Offloads processing from your origin servers.

Cons:
• **Tooling Maturity:** Edge-side composition is still evolving and might lack robust tooling.
• **Complexity:** Managing deployments and debugging can be more difficult when using edge services.

In Summary

- **Iframes** give excellent isolation but at the cost of performance and integration complexity.
- **Client-side approaches** (whether via plain JS orchestration, web components, dedicated frameworks, or module federation) offer a more seamless user experience and allow independent deployment, but they require thoughtful contracts (for mounting, communication, and shared dependencies).
- **Server- and Edge-side compositions** can improve initial load and SEO, but they bring their own challenges in orchestration and deployment.

Choosing the right implementation depends on factors like team structure, performance needs, and how much integration (or isolation) you require between different parts of your application.

cite turn0search2 cite turn0search17

Converting a Next.js App into a Micro-Frontend

Introduction: This report outlines how to turn a React/Next.js application (an AI agent chat interface with session list and team config page) into a single micro-frontend. We compare four client-side integration approaches – global JavaScript mounting functions, Web Components, Single-SPA, and Webpack Module Federation – against the project's needs. Finally, we recommend the best option and provide a step-by-step implementation plan. The goals are easy integration into host React apps, shared state (e.g. auth tokens), high performance with low complexity, and a secure, robust setup.

Evaluating Client-Side Composition Options

Option 1: Global JavaScript Mount/Unmount Functions

Description: The micro-frontend is built as a standalone JS bundle that exposes global functions (e.g. `mount()` and `unmount()`). The host page includes the bundle via a `<script>` tag, then calls `window.MyMicroApp.mount(containerElement, props)` to render the app into a given DOM container. A corresponding `unmount` method cleans up when needed. This is a custom orchestration approach (used by companies like PagerDuty for embedding React apps (How We Use React Embedded Apps | PagerDuty)).

- **Pros:**
  - Simple to implement with plain JavaScript – no special framework required.
  - Integration is straightforward: just include a script and call a global function to mount.
  - Allows passing data/props (e.g. auth tokens, callbacks) directly into the micro-app via function arguments.
  - Minimal overhead in runtime; the micro-frontend runs in the host page context.
- **Cons:**
  - Relies on global variables/functions which can risk naming collisions or accidental misuse. (Using unique names or namespaces mitigates this.)
  - Lacks built-in isolation: the micro-app's HTML/CSS/JS lives in the same global scope as the host, so styling or script conflicts must be managed manually.
  - No standardized interface – it's a custom integration, so each host app must know how to initialize and destroy the micro-frontend.
  - If multiple instances of the micro-app are needed on one page, extra care is required (e.g. unique container IDs, configuring Webpack's `jsonpFunction` to avoid conflicts (How We Use React Embedded Apps | PagerDuty)).

Use Case Fit: This option keeps things lightweight and is easy to drop into any React (or non-React) host. It's effective if you want full control and simplicity, and you can ensure the host and micro-app are kept in sync manually. However, for long-term robustness (especially if multiple hosts or instances are involved), a more standardized approach may be preferable.

Option 2: Web Components (Custom Elements)

Description: Wrap the Next.js app as a **Web Component** – a custom HTML element (e.g. `<ai-agent-app></ai-agent-app>`) that encapsulates the micro-frontend. The micro-frontend bundle registers this custom element via the browser's `customElements.define`. Host applications can then **embed the micro-app by simply adding the custom tag** in their JSX/HTML. The Web Component's implementation uses `ReactDOM.render` internally to mount the React application when the element is inserted into the DOM (and unmounts on removal). This approach is framework-agnostic and leverages native browser capabilities for composition.

- **Pros:**
  - **Easy integration:** Host apps just include the micro-frontend's script (to register the element) and use the custom element in their code. No complex setup – "render the `<my-child-app>` tag and load the script bundle" (How to Create Microfrontends with Web Components in React).
  - **Encapsulation:** The component can attach a Shadow DOM, isolating its CSS/DOM. This prevents style leakage and conflicts with the host app's styles (How to Create Microfrontends with Web Components in React). The Web Component also manages its own lifecycle (connected/disconnected callbacks), improving robustness (auto-unmount on removal) (How to Create Microfrontends with Web Components in React).
  - **Framework agnostic:** Works with any host framework (React, Angular, plain HTML) since it's just a HTML element. This future-proofs the integration.
  - **State and API flexibility:** Properties/attributes on the element can be used to pass data. We can allow "props" by defining element attributes or JS properties and reacting to changes. Hosts can also listen for Custom Events dispatched from the micro-frontend for callbacks or state updates.
- **Cons:**
  - **Learning curve:** Developers must understand how to interact with the custom element (setting attributes vs. properties) and handle events for complex data. It's a slightly different paradigm than a typical React-to-React prop passing.
  - **Data passing limitations:** HTML attributes are string-valued, so passing complex objects (like an auth token object or callbacks) requires using element properties or a custom API. This can be addressed by exposing JS setter methods or using Custom Events for communication.
  - **Duplicate dependencies:** By default, the micro-frontend bundle will include its own React (and other libraries), which could mean the host and micro-app load the same libraries twice. This impacts performance if not handled. (However, this can be mitigated by configuring bundler externals or Module Federation to share dependencies, as discussed below.)
  - **Browser support:** Modern browsers fully support Web Components. Older legacy browsers (IE11) would need polyfills. In 2025 this is usually not an issue, but it's a consideration for broad enterprise environments.

Use Case Fit: Web Components are a strong choice given the project's goals. They offer an easy, **iframe-free** embedding with good isolation and a clear interface boundary. The host and micro-frontend can share state through well-defined element properties/events. This approach keeps the integration **simple and standardized**, improving security (encapsulated scope) and minimizing the risk of integration bugs. It aligns well with a single micro-frontend scenario – the custom element acts as a pluggable widget that any app can insert.

Option 3: Single-SPA (Micro-Frontend Orchestration Framework)

Description: **Single-SPA** is a specialized framework to orchestrate multiple micro-frontends on a page. It acts as a **"shell" application that registers and mounts micro-apps** based on route or other conditions (Module Federation vs Single-SPA | Bits and Pieces). Each micro-frontend is built as a standalone app (which can be React, Angular, etc.), and Single-SPA manages their lifecycles (bootstrap, mount, unmount). The host app would run the Single-SPA "router", and the Next.js micro-app would

need to be adapted as a Single-SPA module.

- **Pros:**
  - **Powerful orchestration:** Handles multiple micro-frontends, enabling complex scenarios (different frameworks or teams for different parts of the UI) (Module Federation vs Single-SPA | Bits and Pieces). It can mount/unmount micro-apps on the fly as the user navigates, improving initial load performance by only loading what's needed (Module Federation vs Single-SPA | Bits and Pieces).
  - **Flexible tech stack:** Because each micro-frontend is isolated, Single-SPA allows mixing frameworks (React for one micro-app, Vue or Angular for another) in one host shell (Module Federation vs Single-SPA | Bits and Pieces). This is useful for gradual migrations or multi-team, multi-tech situations.
  - **Standard lifecycle:** Single-SPA defines a clear API (lifecyle functions) for micro-apps. If the Next.js app is refactored to export `bootstrap/mount/unmount` functions, the integration into the shell is standardized.
  - **Active ecosystem:** There are plugins and community solutions for common issues (routing adaptations, shared global event bus for state, etc.). It's a well-tested approach for enterprise micro-frontends.
- **Cons:**
  - **High complexity/overhead for a single micro-app:** Single-SPA shines when coordinating **many** microfrontends. For just one embedded app, introducing Single-SPA adds extra moving parts (a shell application, configuration for routes or activity functions, etc.) with little benefit. It could be over-engineering in this scenario.
  - **Requires host refactoring:** The host React applications would need to become Single-SPA containers. That means integrating the Single-SPA library, registering the micro-frontend, and potentially altering their routing logic to defer to Single-SPA. This is a significant change, whereas other options can be added to hosts with minimal changes.
  - **Next.js compatibility:** Next.js doesn't officially support Single-SPA out of the box (since Next expects to control the page). Adapting a Next app for Single-SPA can be tricky (Microfrontend support not present in nextjs ? | by Dhana Shekar Tontanahal | Medium) (Microfrontend support not present in nextjs ? | by Dhana Shekar Tontanahal | Medium) – for example, ensuring Next's router and Single-SPA's router don't clash, and disabling Next's SSR.
  - **Shared state complexity:** With Single-SPA, sharing state (like an auth token) usually means relying on an external global store or event bus (since each micro-app is fully isolated). This requires additional plumbing (e.g., using `window.customEvent` or a library like RxJS/Zustand at the shell level (Microfrontend support not present in nextjs ? | by Dhana Shekar Tontanahal | Medium)).

Use Case Fit: Given the **single micro-frontend** use case, Single-SPA is likely **not the optimal choice**. It adds complexity and is designed for multiple independent apps or differing tech stacks – which isn't our scenario. All teams are using React, and we only have one micro-app to embed. While Single-SPA is robust, it overshoots the "low overhead, low complexity" goal here. We would favor simpler integration unless we anticipated adding many more microfrontends or non-React ones in the future.

Option 4: Webpack 5 Module Federation

Description: Module Federation is a Webpack 5 feature that allows one application to **dynamically import modules from another bundle at runtime** (Module Federation vs Single-SPA | Bits and Pieces). In a micro-frontend context, we configure the Next.js app as a **remote**, exposing its components or pages. Host React apps are configured as **hosts** that can load the remote module bundle on demand. Essentially, the host can render the micro-frontend as if it were a normal imported React component – but under the hood the code is loaded from the micro-app's build. Module Federation can also share dependencies between host and remote to avoid duplication.

- **Pros:**
  - **Seamless React integration:** The micro-frontend's UI can be consumed as a React component in the host. For example, the host can do `import RemoteChatApp from "aiAgentApp/RemoteAppRoot"` and then use `<RemoteChatApp token={token} />` in its JSX. This gives a **native integration** feeling – no need for global scripts or custom element APIs.
  - **Shared dependencies:** Module Federation can be configured to treat React, React-DOM (and other libraries) as **shared singletons**. This means the host and micro-app will **not duplicate** common libraries – they'll use the host's copy or the first loaded copy (Module Federation vs Single-SPA | Bits and Pieces). This addresses performance concerns (only one React loaded) and ensures both parts use the same React context (so context providers or hooks could potentially work across the boundary).
  - **On-demand loading:** The host can lazy-load the micro-frontend bundle when needed (e.g., when the user navigates to the chat interface), improving initial load performance. Module Federation's runtime will fetch the remote bundle only when the code is invoked.
  - **No global namespace pollution:** Unlike the global function approach, MF keeps module scope isolated and uses Webpack's module system. There's no need to attach things to `window`. This reduces risk of collisions and is structurally robust.
- **Cons:**
  - **Build configuration complexity:** Both the micro-frontend and each host application must have their Webpack configs adjusted to set up the Module Federation plugin (defining remotes/exposes, shared modules, etc.). In Next.js, this means using a community plugin like `@module-federation/nextjs-mf` and turning off certain Next optimizations. Host apps (if using CRA or custom Webpack) need to integrate Module Federation as well. This setup requires Webpack knowledge and testing to get right.
  - **Tight coupling to Webpack:** Module Federation only works with bundlers that support it (Webpack 5+, or equivalent in other bundlers). If a host app uses a different build system (Vite, older Webpack, etc.), integration is not straightforward. By 2025 most React apps are on Webpack 5 or have a MF plugin available, but it's a consideration.
  - **Deployment coordination:** The host needs to know the URL of the remote app's bundle (e.g., the `remoteEntry.js` file) and potentially its version. This can be managed via environment variables or a manifest, but it requires coordination – deploying a new micro-frontend version might necessitate updating the host's reference if not using a dynamic URL.
  - **Next.js-specific quirks:** Next.js pages are typically SSR or statically rendered, which doesn't translate directly to MF. We would likely expose a client-only component (disabling SSR for that part). Also, the micro-frontend being a Next app means it has its own router and context – when used inside a host, we must ensure it behaves as expected (probably by treating it as a single-page app). These intricacies add some complexity to the conversion process.

Use Case Fit: Module Federation is a **modern and efficient solution** for our scenario if we can handle the build complexity. It especially shines since all apps are React – we can share code and have smooth integration. This approach would eliminate the iframe and allow true shared state (for instance, both host and micro-app could use a common context or store if configured). It does require more upfront work to configure, but once set up, host apps can import the micro-frontend like any other module, which is very **developer-friendly**. If the team is comfortable tweaking build configs, this option provides a clean and performant result. Security is on par with other in-page approaches (the micro-app code runs in the page context), but the isolation of module scope and the ability to namespace modules adds robustness.

Recommended Approach: Web Components for Simplicity and Encapsulation

Considering the trade-offs and the project's goals, **Web Components (custom elements)** emerge as the most suitable choice. This approach best balances **ease of integration** with **low complexity** and **robustness**:

- **Ease of Integration:** Host React apps can use the micro-frontend by simply including a script and placing a `<ai-agent-app>` (for example) in their JSX. There's no need for extensive host reconfiguration or new frameworks. This is as easy as the current iframe approach, but without the overhead of an iframe (no separate context or duplication). The Next.js micro-app can be consumed by any host with minimal effort.
- **Shared State:** We can design the Web Component to accept an auth token and other necessary state as properties/attributes. For instance, the host can do `<ai-agent-app token={authToken} />` in JSX (with slight adjustments to pass a non-string property). The micro-frontend can also dispatch events (e.g., `loginExpired` or `sessionSelected`) that the host can listen to. This two-way communication is straightforward to implement. While Module Federation also allows prop passing (since it's just a React component), Web Components achieve it with standard browser APIs without coupling build systems.
- **Performance:** The Web Component bundle can be optimized to avoid heavy duplication. We have the flexibility to externalize React and others if desired – for example, host apps could load React from a CDN as a global, and the micro-frontend can use `React` from `window` (configured via Webpack externals) (reactjs - Micro-Frontends, Web Components, and Sharing Libraries - Stack Overflow) (reactjs - Micro-Frontends, Web Components, and Sharing Libraries - Stack Overflow). If coordinating that is too fragile, the micro-frontend can include its own React copy – which in practice may be acceptable for now (modern bundlers and HTTP/2 can handle one extra 40KB gzip file, especially if cached). Importantly, with a Web Component we load the micro-app assets on demand (only when the host page that uses it is rendered, similar to code-splitting). This meets performance needs without a complex orchestrator.
- **Security & Robustness:** By using a Shadow DOM in the custom element, we ensure style encapsulation (the host page won't accidentally override the micro-app's styles and vice versa). The boundary also makes the integration more robust – the micro-app can be developed and tested in isolation, as it just needs to render within its own element. There is no heavy runtime coupling with the host, reducing the chances that a host app change breaks the micro-frontend. Additionally, there's no global function to potentially misuse; the interface is constrained to the custom element's API.
- **Lower Complexity than Alternatives:** Single-SPA is overkill here, and Module Federation, while powerful, would introduce build-time complexity for every host. The global function approach is simple but lacks the structural benefits (and could lead to messy integrations down the road). Web Components hit a sweet spot: they leverage web platform standards, requiring only a bit of upfront work to wrap our app into a custom element. Once that's done, using the micro-frontend is as plug-and-play as including an image or a third-party widget.

Note: Module Federation was a close second choice – if all host apps were under unified control and performance (especially avoiding duplicate React instances) was a paramount concern, MF is very attractive. However, given the emphasis on **simple setup for external teams** and broad compatibility, Web Components get the nod. We can mitigate the duplicate dependency issue through build tweaks, and still reap the integration simplicity. In summary, a custom element provides an **easy, secure, and modular integration** path for our Next.js app across various hosts.

Step-by-Step Plan: Converting the Next.js App into a Micro-Frontend (Web Component)

Now we outline a detailed plan to implement the chosen approach. The high-level idea is to **refactor the Next.js application** into a bundle that registers a custom element, which in turn mounts our React app. We'll ensure the app runs client-side only (since host apps will handle overall rendering), and set up smooth communication for state sharing.

1. **Prepare the Next.js App for Client-Only Rendering**
    - **Disable SSR:** Since the micro-frontend will be inserted into an already rendered host page, server-side rendering is not needed (and would complicate integration). Ensure all Next.js pages are either static or client-only. For example, you might remove `getServerSideProps` or `getInitialProps` usages and replace them with client data fetching. If any page needs pre-loaded data, consider using an API call on component mount instead.
    - **Single-Page Mode:** Consider consolidating the Next.js app into one "micro-app" page if possible. You mentioned two pages (chat interface and team config). In a micro-frontend context, it might be easier to treat these as two views within one React app, rather than separate Next.js pages. This avoids Next's page routing interfering with the host. You can use React state or a client-side router (like `react-router` or even Next's own router in CSR mode) to toggle between the chat and config views inside the micro-frontend.
    - **Verify Routing Isolation:** If you do keep multiple Next.js pages, configure Next's `basePath` or `assetPrefix` so that navigation within the micro-app is isolated (for example, using a hash route or a distinct path segment). We don't want the micro-frontend changing the host's URL or doing a full page refresh. One simple approach is to avoid actual URL navigation – instead, manage the "page" switch internally (e.g., show the config UI via conditional rendering or a modal, rather than a new URL).
2. **Set Up the Build to Output a Micro-Frontend Bundle**
    - **Custom Webpack Config:** Next.js allows extending webpack via `next.config.js`. We will use this to ensure our output can run as an embedded script. For instance, set `output.library` and `output.libraryTarget` to define a global name if needed (though with Web Components we might not even need a global, since the act of defining the custom element is the integration point). Ensure the bundle is self-executing or easily importable.
    - **Bundle as a Library (Optional):** You might consider using a tool like `@module-federation/nextjs-mf` plugin or Next's built-in support for "standalone" output. In this case, a simpler route is to compile the Next app as a single bundle. If Next's regular build produces multiple chunks (it will, for each page and for commons), you can still use it – but you'll need to load those chunks in the host. Another approach is to create a separate entry point for the micro-frontend: for example, add a file `src/entry-mfe.js` that imports `ReactDOM` and your root component, then registers the web component (next step). Configure Webpack to treat this as an entry (using Next's `webpack(config)` override). This way, you get a single JS file (plus maybe a CSS file) that includes everything needed for the micro-app.
    - **Externalize Dependencies (Optional):** For performance, decide if you want to externalize React/ReactDOM. This means not bundling React in the micro-frontend. You'd add something like:

      ```
      // next.config.js - inside webpack config override
      config.externals = { react: "React", "react-dom": "ReactDOM" }
      ```

      and ensure the host page includes `<script>` tags to load React and ReactDOM (perhaps from CDN) **before** the micro-frontend script (reactjs - Micro-Frontends, Web Components, and Sharing Libraries - Stack Overflow) (reactjs - Micro-Frontends, Web Components, and Sharing Libraries - Stack Overflow). This allows the micro-frontend to reuse the host's React. If coordinating externals is too risky, you can skip this – the bundle will include React and work independently (at cost of ~additional KBs).
3. **Create the Custom Element Wrapper**
    - **Write the Web Component Class:** In your Next app (or the new entry file), create a class that extends `HTMLElement`. For example:

```
class AiAgentAppElement extends HTMLElement {
  connectedCallback() {
    // This is called when the element is added to the DOM
    const mountPoint = this.attachShadow({ mode: 'open' });  // use Shadow DOM for encapsulation
    const props = {
      token: this.getAttribute('token'), // example of reading an attribute
      // ... you can parse other attributes here
    };
    ReactDOM.render(<App {...props} />, mountPoint);
  }
  disconnectedCallback() {
    // Cleanup when removed
    ReactDOM.unmountComponentAtNode(this.shadowRoot || this);
  }
  static get observedAttributes() {
    return ['token']; // watch token attribute for changes
  }
  attributeChangedCallback(name, oldVal, newVal) {
    if (name === 'token' && this.shadowRoot) {
      // If token prop changes, re-render with new token (or handle as needed)
      const props = { token: newVal };
      ReactDOM.render(<App {...props} />, this.shadowRoot);
    }
  }
}
customElements.define('ai-agent-app', AiAgentAppElement);
```

In this code, `<App />` is your Next.js application's root component (you might import your `pages/index.js` or a custom `<MainApp />` component that includes the chat interface and possibly routing to the config page). We attach a Shadow DOM to encapsulate styles. The `token` attribute is observed so that if the host updates it, we can re-render the app with the new token. (If the token is only set once, you might not need to handle updates; if it might change or refresh, this ensures the micro-app updates accordingly.)

- **Adapt to Next's structure:** If your Next app relies on the default `<App>` in `_app.js`, ensure that your `<App />` component wraps the content appropriately (it might include context providers, etc.). Essentially, we are bypassing Next's usual page mounting and doing it manually. You might import things like Next's `<Head>` or use the same providers as in `_app.js`. The goal is to reuse as much existing code as possible.
- **Inject Styles:** When using Shadow DOM, global styles won't penetrate. If your Next app uses a global CSS or CSS Modules, make sure those styles are included in the bundle and apply within the shadow root. One way is to gather all your CSS into a single file and programmatically attach it to the shadow root: e.g.,

```
const styleTag = document.createElement('style');
styleTag.textContent = /* your compiled CSS as string */;
this.shadowRoot.appendChild(styleTag);
```

Alternatively, if using CSS-in-JS or CSS Modules, those might automatically scope or inject styles; just test that the UI renders correctly when embedded. If not using Shadow DOM, the styles will apply globally — which is easier but sacrifices isolation (choose based on your needs).

- **Testing this component:** You can test this custom element in isolation by creating a simple HTML page, including React, ReactDOM (if externals) and your built JS, then `<ai-agent-app token="test123"></ai-agent-app>`. It should render the app. This ensures your micro-frontend bundle works as a standalone script.

4. **Build and Publish the Micro-Frontend Assets**

   - Run `next build` (or whatever build command now produces your micro-frontend bundle). Verify that the output includes the expected files. Likely you will have `./.next/static/chunks/` containing your script (and possibly a `remoteEntry.js` if using Module Federation, which in this plan we are not, or some custom bundle file from your entry). You may need to locate the main bundle file (for example `main.js` or the file corresponding to `entry-mfe.js` if you set that up).
   - Host these files in a static hosting service or CDN. Essentially, you need a URL from which host apps can load the micro-frontend's script. For security and cache busting, ensure you have a versioning or deployment strategy (you could name the script file with a version or hash, or use the Next.js build output which already includes content hashes in filenames).
   - Also publish documentation for host teams: e.g., "Include this script tag `<script src="https://cdn.yourdomain.com/ai-agent-app/v1.0/ai-agent-app.js" defer></script>` in your index.html (or dynamically import it), then you can use the `<ai-agent-app>` tag." Provide details on available attributes (like `token`) and events the micro-app might emit.

5. **Integrate the Micro-Frontend into Host React Apps**

   - **Include the Script:** In each host application, include the micro-frontend's JavaScript. In a React app, a convenient place is in the public `index.html` (so it loads on all pages). Alternatively, load it on specific pages where needed using a dynamic `<script>` injection or a lazy import of the module (if you expose it as an ES module). The simplest is a script tag for now.
   - **Use the Custom Element in JSX:** Wherever the host wants to render the AI agent interface, use the custom element. For example, in a React component you might write:

```
function AIChatSection({ token }) {
  // ensure the custom element script is loaded before this renders
  return <ai-agent-app token={token} />;
}
```

React will treat `<ai-agent-app>` as an unknown DOM element and render it as-is. The `token` prop here will become a `token` attribute on the element (as a string). If the token is not naturally a string (e.g., it's an object or needs security), convert it (perhaps JWT or session ID string). If you need to pass complex objects or functions, you can get a ref to the DOM node and assign to its property:

```
const agentRef = useRef();
useEffect(() => {
  if (agentRef.current) {
    agentRef.current.someProperty = complexObject;
  }
}, [complexObject]);
<ai-agent-app ref={agentRef}></ai-agent-app>
```

However, try to design the micro-app API to avoid needing this – stick to simple string attributes or a known JSON config object that can be passed as a string if necessary.

- **Shared Context:** For auth tokens, a common pattern is to have the micro-frontend read from a shared source like `localStorage` or a global JS object if both host and micro-app are on the same domain. But since we can directly set a token attribute, we have an explicit pathway that's fine. Just ensure the token is stored in a safe way (e.g., if it's a JWT and the micro-app calls APIs, it can use the token it received).

6. **Implement Shared State & Event Handling**

   - **Passing Data In:** We already handle the auth token via an attribute. You can extend this to any other initial data the micro-app needs (e.g. user info, initial session list). If many pieces of data, you could have one attribute like `config='{"user":"...","theme":"..."}'` as a JSON string that the Web Component parses. Keep it simple and document how to use it.

   - **Emitting Events Out:** Decide what interactions in the micro-frontend should notify the host. For example, if the user logs out or an error occurs in the chat, maybe the host needs to know. In those cases, have the micro-frontend dispatch a CustomEvent. Inside your Web Component's React app, you can call `window.dispatchEvent` or better `this.dispatchEvent` on the custom element. For instance, in React you might get a reference to the DOM node (perhaps via a React portal or by targeting `parentElement` if not using shadow) and do `parentElem.dispatchEvent(new CustomEvent('sessionSelected', { detail: sessionId }))`. Alternatively, since our React app is mounted in the Shadow DOM, we could pass down a callback prop that calls `customElements.get('ai-agent-app')!.dispatchEvent(...)`, but an easier way is to use the CustomEvent from within the Web Component class itself. For example, add a method on `AiAgentAppElement` class:

     ```
     selectSession(sessionId) {
       this.dispatchEvent(new CustomEvent('sessionSelected', { detail: { sessionId } }));
     }
     ```

     Then inside your React app code, detect when a session is selected and call `window.document.querySelector('ai-agent-app').selectSession(id)` – or use a React context to call out. You may need to store a reference to the custom element (perhaps via a context or prop). This part is a bit involved, but even simpler: the React app can dispatch an event on `window` and the Web Component can listen and re-dispatch to host. Design an approach that is least confusing. The host app can listen like:

     ```
     document.querySelector('ai-agent-app')?.addEventListener('sessionSelected', e => {
       console.log("User selected session", e.detail.sessionId);
     });
     ```

     In React, you'd attach the listener to a ref of the element in an useEffect.

   - **Shared Auth:** If the micro-frontend needs to make API calls with auth, ensure the token is passed and used. Since the host likely already handles authentication, the micro-app should not have its own login – it should rely on the token provided. Also consider token refresh: if the host rotates tokens, you'd update the attribute or property so the micro-app can update internal state (our attributeChangedCallback covers that). If using cookies for auth, and the micro-app requests are to the same domain, it might not even need the token explicitly (it could rely on cookies), but passing it explicitly is cleaner.

7. **Ensure Security and Stability**

   - Because the micro-frontend runs in the host page, it has the same privileges as the host code. Standard web security applies (if both are first-party, it's fine). To maintain robustness, add defensive checks in the micro-app: for example, if no token is provided, maybe it shouldn't attempt certain actions. Validate inputs from attributes (since they are strings from the DOM, they could be tampered with if some external site tried to use your component – though if your script is only accessible to trusted hosts, it's less an issue).
   - Implement error boundaries in the React app so that if something goes wrong inside the micro-frontend, it doesn't break the host page. For instance, wrap the app in `React.ErrorBoundary` (if using React 18, the new `errorElement` in routes or a manual boundary component) to catch exceptions and perhaps display a fallback UI within the component. This way, a failure in the micro-app won't propagate an error to the host's React lifecycle.
   - Use Content Security Policy (CSP) on host pages if possible to restrict what the micro-frontend can do (e.g., only allow it to load resources from expected domains). This is more of a general web security measure, but worth noting if the micro-app does dynamic loading.

8. **Testing Integration in Host Apps**

   - Create a simple host (could even be a CodeSandbox or a dummy CRA app) to test the embedded micro-frontend in a realistic scenario. Test passing data in, event communication, and that styles from the micro-app do not bleed out (and host styles don't break the micro-app UI). If you used Shadow DOM, verify that everything is styled correctly inside the shadow. Test in multiple browsers to ensure the custom element works (Chrome, Firefox, Safari – all modern versions support custom elements).
   - Load test and performance: check the bundle size of the micro-frontend. If it seems large, you might revisit splitting or using CDN for some libraries. However, since this micro-app is fairly contained (two main views and some AI logic), it should be manageable.
   - If possible, test with one of the real host applications in a staging environment. E.g., replace the iframe integration with the new micro-frontend and ensure it behaves the same, but now seamlessly embedded. Pay attention to any conflict (for example, if the host app had global CSS that inadvertently styles `<button>` elements everywhere, those might affect the micro-app if not using Shadow DOM – another reason to use Shadow DOM or CSS modules in the micro-app).

9. **Deployment and Rollout**

   - Deploy the micro-frontend bundle to a reliable static hosting (with proper versioning). For instance, version your micro-frontend and allow hosts to embed a specific version script. You might start with a versioned URL (like `/ai-agent-app/1.0/ai-agent-app.js`). Hosts can then upgrade on their schedule by updating the script URL. Alternatively, if you want to centrally update the micro-app for all hosts, you can have them always load a URL like `/latest/ai-agent-app.js` – but this requires coordination to not introduce breaking changes unexpectedly. A middle ground is to use semantic versioning and communicate upgrades to host teams.
   - Update host applications: remove the old iframe integration and add the new script and component usage. Since this is a major change, plan a careful

rollout. You may do it in phases (one host app at a time) to isolate issues. Provide support to those teams since this is a new integration method for them.
- Monitor after deployment: Ensure that the micro-frontend is loading correctly in all hosts. Any errors in console (e.g., custom element not found, attribute issues) should be addressed promptly. With the new approach, if a host forgets to include the script, they'll see a blank component or an unknown element – so having a clear error or warning in place (maybe the Web Component's constructor can log a warning if some required attribute is missing) can help debugging.

10. **Future Enhancements**

    - Once the basic integration is solid, consider enhancements like: **Theming support** (host can pass a theme or CSS variables that your micro-app uses for styling, to better blend into host applications), or **Dynamic import** (host can lazy-load the micro-frontend script only when needed, if not done already). You could also explore using Module Federation in addition to this – for example, to share a library or context, but that adds complexity and might not be necessary if this solution suffices.
    - **Documentation & Maintenance:** Treat the micro-frontend like a mini product. Document its API (attributes, events, any global behaviors). Maintain version control – if you update the micro-app (new features or bug fixes), publish a new version of the bundle and notify host teams to update if needed. Because it's decoupled, you can deploy updates independently, which is a big advantage. Just ensure backward compatibility or communicate breaking changes clearly.

By following these steps, you will have converted your Next.js application into a reusable micro-frontend Web Component. This component can be embedded in multiple React applications seamlessly, allowing shared auth state and a consistent user experience without the silo of an iframe. The integration will be performant (loading only when needed, with no extra iframes or duplicate large libraries) and relatively simple for host apps to adopt. Security is maintained through encapsulation and standard best practices.

Through careful planning, testing, and documentation, this approach sets up a robust micro-frontend architecture for your AI agent interface, meeting the goals of easy integration, shared state, performance, and safety. Enjoy the benefits of micro-frontends with a fraction of the usual complexity!

Sources: The evaluation of approaches and the chosen solution are informed by industry practices and literature on micro-frontend architecture. For instance, PagerDuty's engineering blog describes using global functions for embedding React apps (How We Use React Embedded Apps | PagerDuty), while other experts highlight how Web Components encapsulate micro-frontend functionality and CSS (How to Create Microfrontends with Web Components in React). We also considered insights on Single-SPA's shell approach (Module Federation vs Single-SPA | Bits and Pieces) and Webpack Module Federation's module-sharing capabilities (Module Federation vs Single-SPA | Bits and Pieces) to ensure the chosen path aligns with modern best practices and the specific needs of this project.