ChatGPT

Yjs Bundle Separation in Dual-Mode Architecture

In a dual-mode editor (offline **plain mode** vs online **collaborative mode**), separating the **Yjs** CRDT library from the main bundle can improve initial load performance. Below, we explore the bundle size impact and runtime performance of including Yjs, and discuss key risks of migrating to a lazy-loaded Yjs architecture along with mitigations.

Bundle Impact Analysis (Single-User vs Collaborative Mode)

- **Bundle Size Penalty:** Including Yjs (and its adapters like **y-prosemirror**) in the primary bundle can add significant weight. Yjs's core library is about **93 kB** minified (≈**26 kB** gzipped) [1]. In one case, integrating Yjs with a Tiptap editor extension inflated the bundle by roughly **300 kB** of JavaScript [2] [3], even when collaboration features were unused. This overhead can slow down first-load times, especially on slow networks or devices. Removing Yjs from the default build eliminates this cost for users who never engage collaborative mode.

- **Dynamic Import & Modern Bundlers:** Contemporary bundlers (Webpack 5, Vite, Turbopack, etc.) support **code-splitting** via dynamic `import()`, meaning Yjs can be isolated in a separate chunk that's loaded only when needed [4]. Webpack and Vite will *exclude* Yjs from the main bundle if it's imported dynamically. Each bundler has minor quirks (e.g. Webpack insists on static import paths, Vite pre-bundles ESM modules), but all can defer loading large libraries like CRDTs until runtime. This on-demand loading yields *smaller initial bundles and faster load times*, with one Next.js case study showing a **25% reduction in first-load JS** by using dynamic imports [5]. Turbopack (Next.js' upcoming Rust-based bundler) similarly supports dynamic code splitting and should handle Yjs as a separate on-demand module for the client. The net effect: **users in plain mode download zero Yjs code**, while collaborative users pay the cost only when toggling that feature.

- **Memory Footprint (Static vs Lazy):** When Yjs is bundled statically, the browser must allocate memory for the library code and any initialized Yjs structures on every page load. An empty Yjs document (Y.Doc) itself can consume around **100 KB of memory** just for baseline metadata and structure [6], and Yjs retains tombstone metadata for conflict resolution (about *50% overhead* relative to raw text content) [7]. This means a loaded Yjs editor will use more memory than a plain editor (e.g. a small text note uses ~100 KB, and larger documents incur proportional overhead). By lazy-loading Yjs, you avoid that memory cost entirely for users not in collaborative mode. In other words, **static inclusion = upfront memory + parse cost**, whereas **dynamic inclusion = memory used only when collaboration is activated**. If collaboration is never activated, the Yjs code stays unloaded, keeping the app's baseline memory usage lower.

Runtime Performance Characteristics

- **Initial Render Time:** Removing Yjs from the initial bundle can meaningfully improve first render speed. Every kilobyte of JS has to be fetched, parsed, and executed; by deferring ~25–30 KB gzipped (or more) of Yjs-related code, the browser can render the editor UI faster. In practice, this yields faster **Time-to-Interactive** for plain mode. For example, using dynamic imports in Next.js reduced first-load JS size by over **25%**, directly translating to faster page load times [5]. When Yjs is loaded statically, the browser spends extra milliseconds initializing CRDT structures even if they go unused. In contrast, lazy-loading Yjs only when collaboration starts will introduce a *one-time* delay (during the import) at that moment, but this can be masked with a spinner or preloading hint. Overall, initial loads are snappier without Yjs, improving user experience for the majority of sessions that stay in single-user mode.

- **Memory Usage Patterns:** Loading Yjs has a twofold memory impact: the library code itself and the in-memory CRDT document state. The **code** (~90 KB minified) adds to the JS heap once parsed, and the **Y.Doc** data structure consumes memory proportional to document size and update history. Yjs keeps metadata for conflict-free merging (so a 1 MB document might occupy ~1.5 MB in memory due to CRDT overhead) [7]. If Yjs is loaded but the collaborative features are not actively used, the memory overhead comes mostly from the static footprint. In plain mode (without Yjs loaded), you avoid this entirely. Even an idle Yjs instance (no active edits or peers) will retain some base overhead for the Y.Doc (again, on the order of ~100 KB for an empty doc) [6]. Therefore, **lazy-loading Yjs yields lower memory usage** for users who never enter collaboration, and only incurs the memory cost when needed. When Yjs is unloaded, garbage collection can free that memory if the user leaves collab mode or navigates away (assuming the chunk is no longer referenced).

- **WebSocket Connection Overhead:** In collaborative mode, Yjs usually syncs via a provider (like **y-websocket** or **y-webrtc**) that opens network connections. If collaboration is **disabled or not initiated**, there should be **no WebSocket traffic** at all. When Yjs is dynamically loaded but the app never calls `WebsocketProvider` (or similar), it will not establish any socket connections — meaning no keepalive pings or bandwidth usage. By design, y-websocket sends periodic pings to maintain connections [8], but this occurs only after a client actually connects to a sync server. In an offline/plain session, the WebSocket code is either not loaded or not executed, so it won't consume network resources. This isolation ensures that users in plain mode aren't incurring hidden network overhead. On the flip side, when a user **does** enable collaboration, the runtime cost is the overhead of a WebSocket connection (a few bytes of header for the handshake and small ping messages). This is typically negligible in modern networks, but it's a cost only paid by collaborative users. In summary, **no collaboration = no network overhead**, and when collaboration is active, the socket overhead is minimal (one TCP/WebSocket connection and periodic small messages), which is an acceptable trade-off for real-time syncing.

Risks and Mitigations in Migrating to Lazy-Loaded Yjs

- **Risk:** *Complexity & Bugs When Refactoring Imports* – Changing from static to dynamic imports for Yjs can introduce bugs (e.g. forgetting to load the module before using it, or race conditions in mode switching). **Mitigation:** Audit all places where Yjs is used and guard them behind a clearly defined **feature flag or mode check**. For example, wrap collaborative logic in an `if (isCollabMode) { await import('yjs')… }` block. Use TypeScript to catch mistakes: you can import Yjs types statically (with `import type`) while loading the implementation dynamically, preserving type safety. Write unit tests that simulate both modes to ensure that in plain mode no Yjs APIs are reachable (e.g. assert that a global Yjs object is undefined or that network calls are not made), and in collab mode all needed functionality works after the lazy load. Comprehensive testing (including integration tests where a document toggles into collaboration) will catch sequencing issues early.

- **Risk:** *Performance Regressions on Mode Switch* – Lazy loading introduces a delay when the user first activates collaboration, which could be noticeable (the editor might freeze for a moment while Yjs downloads). **Mitigation:** Use **prefetching and efficient chunk loading**. Modern bundlers allow hinting the browser to prefetch the Yjs chunk *before* the user explicitly needs it. For instance, if a user clicks "Enable Collaboration," you might trigger the dynamic import on the button click, but only actually start collaborative editing once the module is loaded (showing a quick loading indicator). Alternatively, if user roles or settings indicate a high likelihood of collaboration (e.g. a Pro user logged in), you could pre-load Yjs in the background after initial render (using low priority) so it's ready instantly when needed. Ensuring the Yjs bundle is cached (by far-future cache headers on the chunk) also means the cost is paid only once per user. These techniques mitigate UX impact, keeping the mode switch smooth.

- **Risk:** *SEO and Server-Side Rendering Constraints* – If the editor's core content relies on Yjs which is now loaded on the client, could this affect SEO or SSR? In Next.js App Router (React Server Components), for example, you cannot use dynamic imports on the server side. **Mitigation:** Mark the collaborative editor component as a **client-only component** (`"use client"` in Next.js) and dynamically import Yjs only within client-side code. This ensures server rendering isn't blocked or errored by Yjs (which touches browser APIs like WebSocket). For SEO, as long as the page's content (e.g. the document text) is still rendered or hydrated in plain mode, there's no issue – the collaborative features are ancillary. If your application is behind authentication (typical for editors), SEO impact is minimal anyway. Just verify that no critical content is hidden behind the lazy load. Next.js's documentation also notes to **allow dynamic script loading in CSP** if you use a strict Content Security Policy [9] . So, update your CSP `script-src` to include `'self'` (or the CDN domain hosting your chunks) and the `unsafe-eval`/`wasm-unsafe-eval` if needed, so the dynamically loaded Yjs bundle can execute without being blocked.

- **Risk:** *Security and Attack Surface* – Bundling collaboration code by default means any vulnerabilities in Yjs or its network handlers are present on every client, even if unused. By contrast, lazy-loading reduces the exposed surface in plain mode. However, exposing the mode-switch logic might allow attackers to attempt loading the collab module in unintended ways. **Mitigation: Reduce global exposure** of the collaboration logic. For example, do not attach Yjs objects to `window` in plain mode. When splitting the bundle, ensure the collab chunk is only fetched when authorized users trigger it. Even if an attacker manually forces the import, they would still need valid server access to do anything useful. Keep the Yjs provider endpoints secured on the backend (e.g. require auth tokens for the WebSocket server) so unauthorized clients can't simply connect by loading Yjs. From a defense standpoint, the *lazy-loaded approach is safer* — if a zero-day exploit was found in Yjs, it wouldn't affect users who never load it. Monitor security advisories for Yjs and related packages; if a critical fix is released, you may dynamically patch it by updating the chunk without forcing all users to update immediately (since only collab users load it). Lastly, review your Content Security Policy regarding WebSocket and WebRTC: include the collab server's URL in `connect-src` if needed (to allow connecting), and consider that WebRTC peer connections are not directly governed by CSP (browsers may allow them by default). If using WebRTC (y-webrtc), ensure signaling servers are restricted. Proper CSP and feature policy settings can limit where scripts (and by extension, any dynamic loaded code) can connect, mitigating exfiltration risks.

- **Risk:** *Maintenance & Build Complexity* – Creating a dual bundle (plain vs collab) can complicate the build and CI/CD pipeline. You need to ensure the dynamic chunk is built and deployed, and avoid accidental regressions (like someone importing Yjs statically in new code). **Mitigation:** Leverage your build tool's analysis and CI checks. For example, run bundle analyzers in CI to confirm Yjs stays out of the main bundle. You could enforce a **lint rule or import barrier** that disallows `import 'yjs'` in core modules (except in the designated dynamic loader module). For QA, build the app in both modes if possible: one way is to have a feature flag that forces collaboration mode on in a staging build, so you can test the collab bundle in isolation. Also document this architecture for your team so future developers understand the pattern. If using Module Federation or Nx/Turborepo, you might consider separate entry points for collab – but often a single app with dynamic import is simpler. The key is automated tests: have end-to-end tests that run the app with collaboration enabled to exercise the Yjs code path, and others that run with it disabled to ensure nothing loads. In CI, you can even verify the network requests (e.g. using Playwright/Puppeteer) to confirm that in plain mode, the Yjs chunk file and WebSocket calls never occur. This guards against regressions and gives confidence in the separation.

By addressing bundle bloat upfront (lazy-loading ~30 KB+ of Yjs only when needed) and being mindful of performance and security implications, you can achieve a **leaner, faster loading editor for single-user cases** while still providing robust collaborative features on demand. A careful migration with feature flags, thorough testing, and observance of best practices (in bundling, CSP, and network security) will minimize risk and ensure a smooth transition to a dual-mode architecture [10]. The result should be concrete gains: a smaller core bundle (no Yjs overhead for non-collaborative users), minimal impact on collab users (with optimized loading), and a clear **separation of offline vs online mode** functionality for easier maintenance and potential future enhancements.

Sources: Yjs bundle size data [1] [3]; Webpack code-splitting guide [4]; Next.js dynamic import performance [5]; Yjs memory and overhead discussion [6] [7]; Next.js CSP recommendations [9]; Tiptap collaboration mode request [10].

---

[1] Best of JS • Yjs
https://bestofjs.org/projects/yjs

[2] [3] [10] Reduce bundle size of `extension-drag-handle` · ueberdosis tiptap · Discussion #5787 · GitHub
https://github.com/ueberdosis/tiptap/discussions/5787

[4] Code Splitting | webpack
https://webpack.js.org/guides/code-splitting/

[5] 25.33% Reduction in First Load JS with NextJS Dynamic Imports | Kevin Wang's Blog
https://thekevinwang.com/2021/03/15/reduce-first-load-js

[6] [7] Scalability of y-websocket server - Yjs Community
https://discuss.yjs.dev/t/scalability-of-y-websocket-server/274

[8] Disconnects and reconnects every 30 seconds · Issue #24 - GitHub
https://github.com/yjs/y-websocket/issues/24

[9] How to set a Content Security Policy (CSP) for your Next.js application
https://nextjs.org/docs/app/guides/content-security-policy