The honest case for and against sending HTML from the server

Server-rendered HTML via the hypermedia approach (HTMX and its philosophical siblings) is architecturally sound for roughly 80–90% of web applications, but hits hard, physics-based walls for the remaining 10–20%. The strongest evidence comes from production migrations showing **67% code reductions** Htmx +4 and **50–60% faster first loads**, alongside equally credible reports of developers hitting "explosions of complexity" when pushing HTMX beyond its sweet spot. Hacker News The debate isn't really about which approach is "right" — it's about where the line sits between server-appropriate and client-appropriate interactivity. The industry is converging on a hybrid "transitional" model, but HTMX advocates and SPA advocates draw that line in very different places.

The 48-essay philosophical foundation at htmx.org

The htmx project maintains **48 essays** (44 current, 4 legacy from intercooler.js) that constitute the most thorough articulation of the hypermedia position in modern web development. These cluster into six themes: hypermedia theory and HATEOAS (10 essays), REST critiques (7), SPA criticism and case studies (9), server-side architecture philosophy (5), project philosophy (7), and software engineering philosophy (6).

The foundational argument runs as follows. Roy Fielding defined REST in 2000 as the web's architecture: HTML exchanged over HTTP, with **hypermedia as the engine of application state** (HATEOAS). htmx The industry hijacked the term — first for XML/SOAP APIs, then for JSON APIs — despite neither being natural hypermedia. htmx Carson Gross's essay "How Did REST Come To Mean The Opposite of REST?" traces the path: InfoWorld REST was the opposite of SOAP → JSON APIs weren't SOAP → therefore JSON APIs became "REST." htmx The htmx team argues this inversion is not pedantic — it caused the industry to abandon the architectural properties that made the web work. InfoWorld

The practical consequence is the **Hypermedia-Driven Application (HDA)** architecture, which the essays position as "thesis: MPA, antithesis: SPA, synthesis: HDA." htmx HDAs use declarative HTML-embedded syntax for interactivity, exchange HTML (not JSON) with the server, and keep scripting as enhancement rather than infrastructure. htmxhtmx The essay "When Should You Use Hypermedia?" provides the most honest self-assessment: hypermedia excels for text/image-heavy UIs, CRUD applications, and nested UIs with well-defined update areas. htmx It struggles with dynamic interdependencies spanning screen regions (spreadsheets), offline requirements, and high-frequency interactions htmx (games, maps). htmx +2 The essay

explicitly states that **Twitter and Gmail could be implemented in hypermedia, but Google Sheets and Figma cannot**.

Four real-world migration case studies — React-to-HTMX, Next.js-to-HTMX, and WASM-to-HTMX — provide the empirical backbone. The flagship Contexte case documented a **67% codebase reduction** Htmx (21,500 to 7,200 LOC), htmx **96% fewer JavaScript dependencies** (255 to 9), and **50–60% faster first loads** with no UX regression. The entire team became full-stack. htmxhtmx Gross notes these results partly reflect Contexte's content-focused app being ideal for hypermedia. htmxhtmx

---

Software bloat validates the hypermedia instinct

The IEEE Spectrum article "Why Bloat Is Still Software's Biggest Vulnerability" by Bert Hubert (February 2024) provides independent ammunition for the server-rendered position. Hubert's core thesis: **software bloat is the primary driver of software insecurity**, and the more code deployed, the more opportunities for exploitable vulnerabilities — regardless of code quality. IEEE Spectrum

His data points are striking. A typical Electron JS app incorporates Chromium and Node.js, entailing an estimated **50+ million lines of code**. IEEE Spectrum A simple garage-door-opener app requires over 50 million active lines of code running across several OS images on multiple servers. ieee One photo-sharing app on GitHub imports **1,600 external code libraries** of unknown provenance. AskWoody Hubert's proof of concept, Trifecta (a minimalist image-sharing app), achieved equivalent functionality in **1,600 lines of code with ~5 dependencies and 3 MB total size** — compared to a competing Node-based solution with 1,600 dependencies totaling 4+ million lines of JavaScript shipped as a 288 MB Docker image. ieee

Hubert's framing reinforces the htmx team's "complexity budget" argument: every dependency and abstraction layer adds attack surface and maintenance burden. The EU has responded with three pieces of legislation (NIS2, Cyber Resilience Act, Product Liability Directive) to force vendors to minimize attack surfaces. ieeeIEEE Spectrum The philosophical alignment between Hubert's lean-software argument and the hypermedia approach is clear — HTMX's **14 KB single-file, zero-dependency architecture** Htmx +3 is the antithesis of the Electron model.

Where HTMX genuinely breaks down: five hard limits

Extensive community research reveals five categories where HTMX hits architectural walls that no amount of extensions or clever patterns can fully resolve.

1. High-frequency client-side interactions are physically impossible. Canvas-based tools, drawing apps, games, and anything requiring sub-50ms feedback loops cannot tolerate network round-trips. The htmx documentation explicitly acknowledges this: "For high-frequency interactions (real-time drawing apps, fast-paced games), the network round-trip is unacceptable." Refactor This is not a framework limitation — it's a physics constraint.

2. Complex interdependent client-side state defeats the server-render model. Google Sheets is the canonical example: a user can enter an arbitrary formula into a cell and introduce dependencies that cascade across the screen dynamically. htmx Issuing a server request on every cell update is untenable. htmxHtmx Collaborative editors requiring CRDTs or operational transformation algorithms are similarly incompatible — these are pure client-side computation problems.

3. Multi-location UI updates from single state changes create wiring nightmares. The developer known as swyx (prominent frontend engineer) documented this precisely: adding a count of todos on each tab that updates on add/edit/delete requires coordinating `hx-swap-oob` across multiple elements — essentially manual wiring that doesn't scale. In React, this is one state update. He concluded: *"This is htmx's explosion of complexity that makes it not optimized for change."* Hacker News

4. DOM state preservation remains fundamentally fragile. Chris Done's detailed production critique documented how replacing `outerHTML` destroys browser-local state — open/closed `<details>` elements, `<input>` values, scroll positions, dropdown states. chrisdone The idiomorph/morphdom extensions mitigate this but introduce new edge cases: HTMX stores its request queue on DOM elements, and morphing (which preserves elements rather than replacing them) creates "undefined behavior" where design assumptions are violated. chrisdone Done's team had to **patch morphdom** to avoid overwriting input and details elements.

5. Offline capability is essentially zero. The htmx team states plainly: "If your application requires full functionality in an offline environment, then the hypermedia approach is not going to be an acceptable one." htmx Service Workers are acknowledged as "a complex option" but not a practical solution for full offline functionality. htmx

Beyond these hard limits, three softer constraints matter in practice. The **component library ecosystem gap** is significant — developers repeatedly cite ShadCN, Ant Design, and similar React component libraries as major productivity multipliers that have no HTMX equivalent. htmxhamy The **AI tooling gap** was cited by Gumroad when they chose Next.js over HTMX: "AI tools are intimately familiar with Next.js and not so much with htmx." htmx And **hiring** remains a practical concern — as one critic noted, "there are, rounding off, zero htmx jobs." htmx

---

Where people haven't tried hard enough

The research also reveals scenarios where HTMX's limits are more perceived than real. **Dashboard-style applications** are buildable with SortableJS for drag-and-drop interactions (HTMX handles the persistence side via POST), SSE extensions for live data updates, and Alpine.js for client-side toggle states. Zorro.management demonstrates this at scale with **1,200+ hx- attributes** and custom JS for drag-and-drop and mood boards. GitHub RisingStack modernized a **50-million-unique-visitors/month streaming platform** using HTMX + Alpine.js + Python Flask. risingstack

The **islands of interactivity pattern** is the key architectural insight enabling complex HTMX applications. HTMX handles ~90% of interactions (the server-driven majority), while Alpine.js, Web Components, or targeted JavaScript handle the ~10% requiring rich client-side logic. Loren Stewart Communication happens via browser events. As one practitioner documented: "HTMX covers about 90% of our client-side needs. For the remaining 10%, we turn to Lit to build standard, performant Web Components." HTMX 2.0 added dramatically improved Web Component support including Shadow DOM processing. htmxGitHub

Carson Gross himself endorses this pragmatic hybrid: *"Do not be dogmatic about using hypermedia. At the end of the day, it is just another technology with its own strengths & weaknesses. If a particular part of an app demands something more interactive than what hypermedia can deliver, go with a technology that can."* htmxhtmx

The Alpine.js complement fills specific gaps: `x-show` / `x-if` for showing/hiding elements without server round-trips, client-side filtering and sorting of already-loaded data, `x-transition` for enter/leave animations, and lifecycle hooks for wiring up third-party libraries. Alpine's MutationObserver automatically detects HTMX DOM swaps and re-initializes Alpine components. risingstack However, even the HTMX + Alpine.js combo falls short for complex client-side state synchronization across components, offline-first applications, and highly interactive interfaces like collaborative editors.

Server-side frameworks that offer more than HTMX

Six frameworks maintain state primarily server-side while offering varying degrees of client-side power beyond HTMX's capabilities.

Phoenix LiveView (Elixir) represents the highest ceiling among server-side frameworks. Each connected user is backed by a lightweight BEAM process maintaining state. Curiosum The WebSocket connection sends only **minimal diffs** (static/dynamic parts separation) rather than full HTML fragments. Aitor Alonso At roughly **40 KB memory per connection**, a 1GB server can host ~25,000 concurrent LiveViews, InfoQ leveraging the Erlang VM's famous ability to handle

millions of lightweight processes. Cars.com and Fly.io use LiveView in production Piccalilli for real-time dashboards. The tradeoff: Elixir-only, no offline capability, and WebSocket backpressure under load can cause lag if event handlers are expensive. Medium

Hotwire/Turbo (Rails) is architecturally closest to HTMX — stateless HTTP with optional WebSocket push via Turbo Streams + ActionCable. Turbo Drive intercepts navigation and replaces the body, Turbo Frames scope updates to regions, and Stimulus handles client-side behavior. At **~45 KB** combined (Turbo + Stimulus) versus HTMX's 14 KB, DEV Community it's more opinionated and Rails-centric but offers first-class WebSocket support. Known friction points include HTML incompatibilities with `<turbo-frame>` custom elements (which break `<table>` content). Js

Laravel Livewire uses a unique hybrid model: state is serialized and sent **with each AJAX request** — the server is stateless between requests but rebuilds component state from the payload each time. Livewire v3 includes Alpine.js in core. Joel Male For complex forms, developers report **500ms–1200ms response times** without optimization Livewire Forum when large Eloquent models are serialized/deserialized. Real-time push requires a separate Laravel Echo + Pusher/WebSocket stack. Fly

Blazor Server (.NET) maintains a persistent SignalR WebSocket connection per "circuit" (user session). Microsoft benchmarks show **5,000+ concurrent users** on a single 1 vCPU/3.5GB VM and **20,000+** on 4 vCPU/14GB, Microsoft Learn with Azure SignalR Service scaling to 100,000 connections. Devready The cost: **~85 KB server memory per connection** (without application data), Devready mandatory sticky sessions, Xafmarin and zero offline capability.

Vaadin (Java) is the most extreme server-side approach — all UI logic runs in Java with no JavaScript knowledge required. Vaadin The entire UI component tree lives in server memory Medium at **50 KB–1 MB per user**. Vaadin It offers the richest built-in component library (enterprise-grade data grids, charts, WCAG 2.1 AA accessible) Vaadin but the poorest scalability Philipp Hauer's Blog — typically hundreds to low thousands of concurrent users per server.

Datastar is the most architecturally interesting newcomer. Created by Delaney Gillilan (also an HTMX contributor), it combines HTMX + Alpine.js functionality into **11.3 KB** — smaller than HTMX alone. Its core innovation is using **Server-Sent Events (SSE)** as the primary communication mechanism rather than AJAX, enabling efficient real-time push natively. Reactive "signals" are kept in the browser but "owned by the server." It's backend-agnostic with SDKs for Go, .NET, Node, PHP, Python, and Rust. Demonstrated capabilities include a One Billion Checkboxes demo and real-time collaborative apps. However, it reached v1.0 only recently and has a small community — a calculated risk for production use.

Framework	State location	Protocol	Memory/user	Max concurrent users/server	UI complexity ceiling
HTMX	Stateless	HTTP	None	Standard HTTP scale	Medium
LiveView	Server process	WebSocket	~40 KB	~25K (1GB)	High
Hotwire/Turbo	Stateless	HTTP + WS	None	Standard HTTP scale	Medium-High
Livewire	Hybrid (serialized)	HTTP/AJAX	None	Standard HTTP scale	Medium
Blazor Server	Server circuit	WebSocket (SignalR)	~85 KB	5K–20K per VM	High
Vaadin	Server session	HTTP/XHR	50KB–1MB	Hundreds–low thousands	Very High
Datastar	Hybrid (signals)	HTTP + SSE	None	Standard HTTP scale	Medium-High

The philosophical core: where to draw the line

The debate between server-rendered HTML and client-side state management is fundamentally about **where to draw the line between server and client responsibility** — not about which approach is universally correct.

The hypermedia position rests on three pillars. First, **most web apps are simpler than we pretend** — the majority of business applications are CRUD-y forms and lists that don't need Google Sheets-level client-side state. Htmx Second, **accidental complexity is real and enormous** — build toolchains, state management libraries, hydration strategies, and client-server serialization represent massive overhead for displaying text and images. htmxhtmx Third, **single source of truth is powerful** — the UI essentially cannot be out of sync with the database because the HTML was just generated from it, Refactor eliminating an entire class of state synchronization bugs.

The strongest counter-arguments also number three. First, **some UIs are inherently client-stateful** — no amount of server rendering elegantly handles spreadsheets, collaborative editors, or gesture-driven interfaces. These require maintaining dynamic interdependencies that can't be determined at server-render time. htmxhtmx Second, **latency is a physics problem** — network round-trips cannot be eliminated, and for latency-sensitive interactions, optimistic UI updates aren't optional but essential. Third, **component state management is unsolved in hypermedia** —

Chris Done's production critique and the Gumroad case study demonstrate that real-world HTMX applications struggle with maintaining state across multiple interactive regions, a problem React's component model handles naturally. chrisdone

Dan Abramov framed the synthesis elegantly: most UIs need **UI = f(data, state)** — both server-side data access and client-side state. Overreacted React Server Components, Astro Islands, and Remix all attempt to let developers compose across this boundary. Overreacted The htmx team's response is that this is overengineered — for most applications, HTML from the server covers the `f(data)` part, and Alpine.js or Web Components handle the `f(state)` islands.

Rich Harris coined "Transitional Applications" as the middle ground. Bram.us Carson Gross "violently agreed" with the concept but argued the line between what needs client-side interactivity versus what can be done with hypermedia is **much further toward hypermedia than Harris suggests**. htmxhtmx This is the crux of the disagreement — not the philosophy, but the practical boundary.

Conclusion: the honest architectural assessment

The evidence supports three conclusions that resist simplification. **First, the hypermedia approach is underrated for typical applications.** Production migrations consistently show 50–67% code reductions, faster loads, and elimination of entire dependency categories. htmxhtmx The Paris 2024 Olympics used HTMX for critical network automation infrastructure. htmx Most web applications — internal tools, content sites, e-commerce, admin panels, CRUD apps — are well within HTMX's sweet spot. htmxhtmx

Second, the hard limits are real and non-negotiable. Offline-first, sub-50ms interactive feedback, complex interdependent client state, and collaborative editing are not problems of insufficient effort — they are architectural incompatibilities with the request-response model. GitNation No extension or pattern eliminates the speed of light.

Third, the industry is converging on hybrids, but from different directions. HTMX starts server-first and adds client islands; React starts client-first and adds server components. Both are moving toward the center. The practical question for any team is not "HTMX or React?" but "what percentage of our UI genuinely requires client-side state?" If the answer is under 20%, the hypermedia approach will likely yield a simpler, more maintainable system. Htmx If the answer is over 50%, a framework with a native component model will prevent the "explosion of complexity" that swyx documented. Hacker News The honest answer for most teams is probably: **start with hypermedia, add islands of JavaScript where physics demands it, and resist the temptation to reach for a SPA framework until you've proven you need one.** htmx