American Academic Scientific Research Journal for Engineering, Technology, and Sciences
ISSN (Print) 2313-4410, ISSN (Online) 2313-4402
https://asrjetsjournal.org/index.php/American_Scientific_Journal/index

Typical Patterns of Interaction between a React Frontend and a WordPress Backend

Karen Sarkisyan*

Senior Software Engineer at EPAM Systems, Inc, Belgrade, Serbia
Email: karen.sarkisyan01@gmail.com

Abstract

This article reviews current practices of using React frontend with WordPress backend in a headless setup and typifies main data-transfer patterns, rendering strategies, and auth/reactivity mechanisms. Massive growth in the headless-CMS market, a leading position for WordPress, and the widespread use of React justifies this study's relevance. The novelty of this work lies in building a three-dimensional model that integrates the data channel (REST vs GraphQL vs RPC) with rendering strategy (CSR, SSR, SSG/ISR) and authorization/update approach (Cookie + Nonce, JWT, Webhooks/Subscriptions), allowing the typical interaction patterns — over ten of them — to be classified and assessed. The significant findings indicate that REST-SPA has a minimal entry threshold due to the built-in WP-REST API but needs more caching to completely get rid of the "N+1" problem and reduce network latency; GraphQL-SPA solves aggregated request problems and also has strict typing but it adds much complexity to schema and access-control design; Next.js Solutions with SSR/ISR have both Static Generation and Incremental Updates via Webhooks or GraphQL Subscriptions. They are high performing, SEO friendly, and offer content consistency; in private scenarios, JWT authorization or request proxying is used; for headless e-commerce, CoCart is chosen; microservice REST-RPC endpoints extend platform capabilities without forking the core. This article will be helpful for architects, developers, and technical leaders choosing an optimal headless infrastructure based on React and WordPress.

Keywords: headless; React; WordPress; REST API; GraphQL; SSR; SSG; ISR; JWT; webhooks.

Received: 5/1/2025
Accepted: 6/22/2025
Published: 7/4/2025

---

* *Corresponding author.*

1. Introduction

The headless approach resolves the age-old conflict between dynamic-content requirements and frontend speed by decoupling the traditional WordPress monolith into an administrative "content-management system" and an autonomous presentation layer. Developers gain the flexibility to choose their tech stack, and editors retain the familiar WP-admin interface; however, the price of freedom is increased integration complexity, the necessity for bidirectional authorization, and the need to readdress SEO tasks previously automated by themes. This balance of benefits and costs makes the study of typical interaction patterns in demand.

The market confirms the maturity of the concept: according to Market Research Future, the global headless-CMS market reached USD 3.26 billion in 2024 and, at an average annual growth rate of 21.4%, will exceed USD 22 billion by 2034 [1]. A WP Engine survey showed that 73% of companies already use headless, and 98% plan to evaluate it within the next 12 months; 60% are budgeting infrastructure increases for these projects [2]. WordPress remains the dominant CMS platform, accounting for 61.2% of sites where the content-management system is known, making it the most frequent backend in headless scenarios [3]. The React + WordPress pairing is driven not by marketing but by development statistics and economics. Last year, 85% of frontend developers used React, and only one in five rated it negatively [4], guaranteeing easy hiring and a vibrant ecosystem of libraries. WordPress, in turn, offers a mature content model, thousands of plugins, and inexpensive hosting; migrating to headless mode does not require abandoning the familiar editorial UX. These factors form a rational business motivation: minimal migration costs, fast time-to-market, and the prospect of scaling via CDN and serverless without recursive reengineering of the content base.

2. Materials and Methodology

The materials and methodology of this study are based on a systematic analysis of 23 sources, including industry reports, developer surveys, plugin and npm package statistics, official documentation of WordPress and React frameworks, and production-deployment case studies. Key sources include: the Market Research Future report on the headless-CMS market [1], the WP Engine survey on headless adoption [2], W3Techs data on WordPress's CMS market share [3], The Software House study on frontend-stack popularity [4], official releases WordPress 4.7 and WPGraphQL [5, 6], Next.js and Apollo Client download metrics from npm [7, 17], Datadog and Cloudflare reviews of serverless infrastructure [8, 9], the Ledger case on Vercel [10], as well as plugins for JWT authorization and REST caching [14, 16] and CoCart usage statistics for headless WooCommerce [21].

The theoretical foundation comprises works on the evolution of WordPress from a monolithic CMS to an API-centric platform [5, 6, 23], studies on headless-architecture practices and their implementation economics [1]–[4], and reviews of modern React-application rendering approaches (CSR, SSR, SSG/ISR) and their impact on user experience [7, 13]. To assess serverless execution technologies and edge extensions, data from Datadog on AWS Lambda and Google Cloud Functions adoption [8] and the Cloudflare Workers report [9] were employed.

Yet the data also reveal persistent asymmetries. Surveys by WP Engine and The Software House indicate that

only a minority of WordPress agencies have transitioned client work to a fully headless model, primarily due to the perceived operational overhead of maintaining two separate stacks [2, 4]. Case studies, such as Ledger's migration to Vercel, indicate that these overheads can be amortized when global latency budgets are tight or when edge personalization is required; however, the tipping point varies sharply by traffic volume and team maturity [10]. Likewise, Datadog's telemetry on cold-start distributions for AWS Lambda suggests that serverless rendering is price-efficient for bursty workloads. In contrast, high-QPS e-commerce stores may still lean on provisioned SSR hosts or incremental static regeneration to guarantee <100 ms TTFB [8].

The analytical model proposed here helps interpret such trade-offs. Projects anchored on REST + CSR can ship quickly and leverage existing WP plugins with minimal friction, but they inherit cookie-based nonce semantics that complicate cross-origin scenarios [5, 12]. GraphQL pivots (WPGraphQL + Apollo) add schema introspection and fine-grained queries, yet they require a JWT or signed-header strategy to remain cache-friendly at the CDN layer [6, 13, 14]. On the rendering axis, ISR has gained traction because it combines CDN-level cache hits with on-demand revalidation hooks, a pattern that Next.js and Vite have begun to abstract behind one-line configuration flags, effectively narrowing the operational gap between Jamstack and traditional server-side rendering (SSR) [7, 17].

The research methodology included three interrelated stages. First, a systematic review and content analysis of official WordPress documentation (REST API, WPGraphQL, register_rest_route) and the React ecosystem (Next.js, Vite, Apollo Client) to identify primary interfaces, query schemas, and caching options [5, 7, 17]. Second, quantitative collection of metrics: CMS market share and adoption statistics (W3Techs, Barn2) [3, 11], weekly npm-package downloads (Next.js, Apollo Client) [7, 17], active plugin installations (WPGraphQL, JWT, CoCart) [5, 14, 21], and usage of serverless solutions and webhook triggers [8, 9]. Third, development of an analytical model comprising three axes—data channel (REST vs. GraphQL vs. RPC), rendering strategy (CSR, SSR, SSG/ISR), and authorization/reactivity mechanism (Cookie + WP Nonce vs. JWT vs. Webhooks/Subscriptions) [12, 13].

Within the framework of the study, primary sources included open industry reports, NPM repository statistics, the WordPress plugin catalog, and official documentation. This documentary foundation provided a representative snapshot of the technologies as of early 2025; however, it did not include the author's load-testing experiments or response-time measurements—their implementation could constitute a promising avenue for further work. Additionally, the sample's time frame is limited to April–May 2025; later versions of WordPress, Next.js, or new caching methods may introduce additional differences that we have not yet reflected in the model. This paper examines the React + WordPress setup as the most common combination in headless project use; a deep dive into other front-end tools or CMSs was not part of this study. The selected authorization options (Cookie + Nonce and JWT) and reactivity mechanisms (Webhooks, experimental GraphQL Subscriptions) are described in their most common form and do not cover exotic scenarios such as multi-factor authentication or Zero-Trust networks.

4. Results and Discussion

As the Introduction outlines, transitioning to a headless architecture is impossible without an API-centric WordPress. In its early years, the platform generated HTML directly via PHP templates, and external integrations were limited to XML-RPC. A qualitative shift occurred in December 2016, when version 4.7 introduced full REST endpoints for posts, taxonomies, users, and settings, officially opening WordPress to external clients [6]. The WPGraphQL plugin drove the next evolution, with its strictly typed schema and custom resolvers, enabling the aggregation of related entities in a single request; active installations of the plugin now exceed 30,000, and the project is moving toward "canonical" status within the core ecosystem [5]. Thus, over eight years, WordPress has progressed from server-rendered themes to an API platform that can confidently serve as the head of any modern frontend framework.

On the client side, React remains the undisputed leader. The classic create-react-app workflow is still popular for internal dashboards, but meta-frameworks dominate in public-facing products. Next.js—now downloaded over 10 million times per week on npm—reflects its true market share and ensures long-term community support [7]. Vite provides rapid local builds and a "hot" developer experience. Paired with WPGraphQL, this combination addresses two critical needs of headless sites: a strict data model and performant rendering on the React side.

The final link in the chain is the global edge infrastructure, where functions run closer to the user and offload work from the PHP backend. Report [8] records that serverless solutions are used by over 70% of organizations on AWS and 60% on Google Cloud, with adoption continuing to grow, as shown in Figure 1.

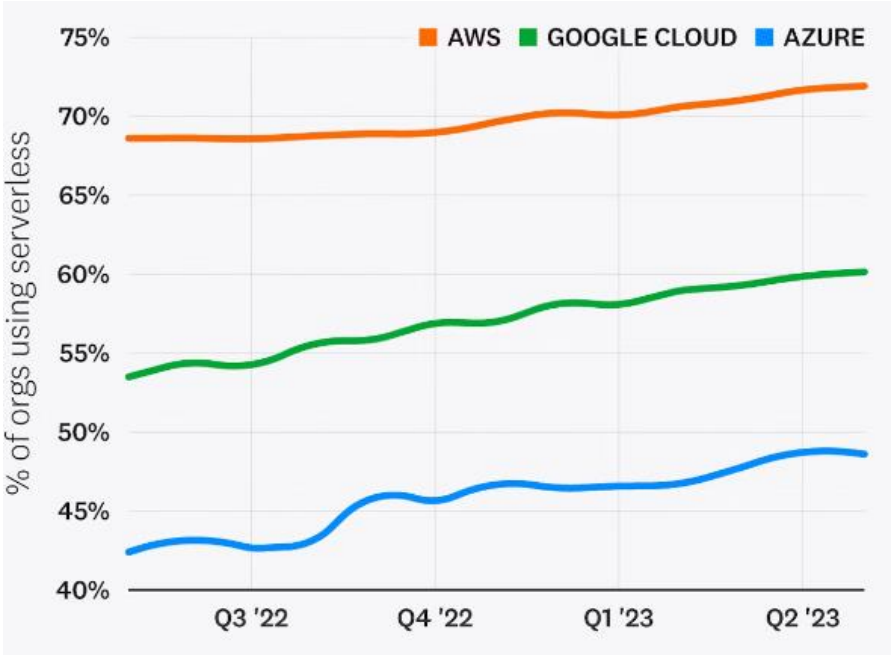

Figure 1: Serverless adoption by cloud provider [8]

Upper-layer platforms are developing even more rapidly: over 2.4 million developers now deploy applications

on Cloudflare Workers, R2, and AI services [9], and the Ledger case demonstrated that moving the Next.js frontend to Vercel Edge Functions yielded a 67% reduction in load time while handling 6–7 million requests per day [10]. For headless WordPress, heavy operations—from ISR validation to authorization Lambda handlers—can be moved to the perimeter, preserving the editorial environment on familiar hosting and sharply reducing TTFB for the end user.

Thus, the evolution of WordPress code, the maturity of the React ecosystem, and the widespread adoption of edge serverless converge at a technological point where typical interaction patterns become not merely best practices but a condition for a site's competitiveness.

Having characterized WordPress's evolution as a move toward API centrism, defining a coordinate grid on which specific integration patterns will subsequently be mapped is logical. The first axis is the method of data delivery. The basic REST layer has been included in the core since version 4.7. According to the report [11], it is thus formally available to all 472 million sites currently running on WordPress, the platform's total. Report [12] showed that WordPress's overall website market share grew from 27.3% in early 2017 to 43.6% in January 2025, as illustrated in Figure 2.

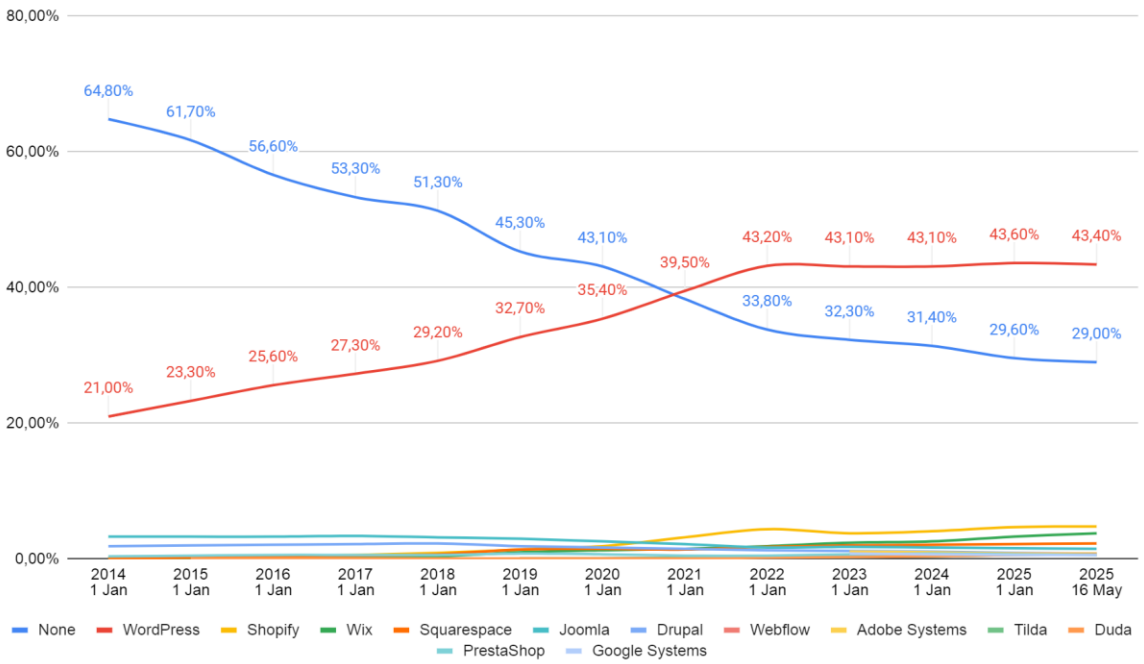

Figure 2: Historical yearly trends in the usage statistics of content management systems [12]

Extending the schema via register_rest_route enables rapid release of RPC-like endpoints, but REST suffers from the "N+1" problem for complex selections. The emergence of WPGraphQL was the response, and Automattic is shepherding it toward canonical status, effectively institutionalizing GraphQL within the WordPress ecosystem [5]. Thus, on the "REST ↔ GraphQL ↔ RPC" scale, a project's position is determined by the balance between ease of implementation and the need for aggregated queries.

The second axis is defined by the choice of rendering strategy. The survey [13] shows that, over the past year, 90% of respondents wrote Single-Page Applications, 59% employed server-side rendering, and 46% experimented with static generation; the total sample comprised 11,119 developers. The full survey results are presented in Figure 3.

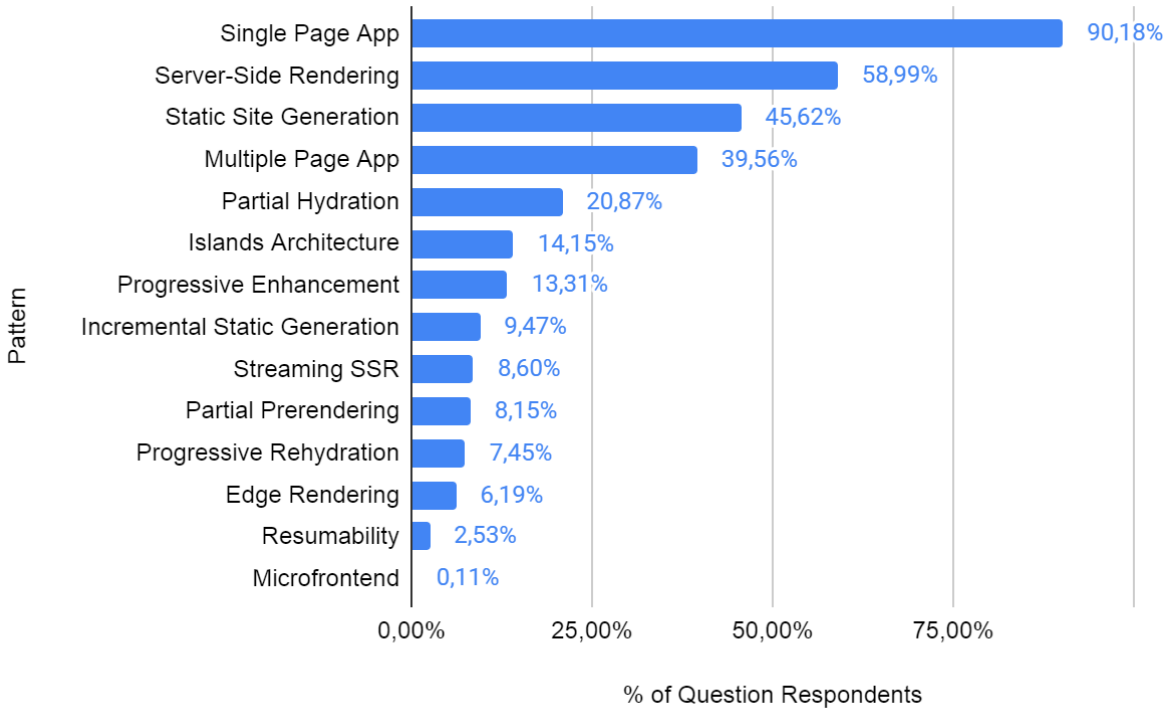

Figure 3: Which architecture and rendering patterns have you used in the last year? [13]

Figures confirm that CSR remains the "default," but SSR and SSG/ISR have already moved from niche techniques into the mainstream. For headless WordPress, this implies three distinct modes: pure SPA—minimal server dependencies; SSR—optimal TTFB and personalization; SSG/ISR—the ideal option for frequently updated yet public pages, where Next.js automatically rebuilds content upon receiving a webhook signal.

The third axis is the combination of authorization and reactivity. The classic Cookie + WP Nonce approach suffices as long as the frontend remains under the backend's domain; otherwise, developers generally have two options. One is a JWT layer, for which a WordPress-repository plugin exists with over 60,000 installations, regularly maintained and supporting refresh tokens [14]. The other is proxying requests through a custom backend-for-frontend, simplifying security but increasing overall latency. When content reactivity is required, the picture is completed by WP Webhooks or experimental GraphQL Subscriptions; these send events to publications and thereby trigger revalidation in the ISR chain. Experience shows that the choice of authorization mechanism often "drives" the reactivity solution: JWT pairs conveniently with WebSockets or SSE, whereas the cookie scheme aligns more naturally with REST and one-time nonce tokens.

Combining the three axes—data channel, rendering method, and authorization/reactivity mechanism—any

headless project can be classified, and its bottlenecks can be predicted at the architectural sketch stage. This forms the basis for selecting a specific React-frontend ↔ WordPress-backend interaction pattern.

The catalog begins with two opposite yet most common patterns, which fit neatly into the previously defined "channel – render – authorization" coordinates.

REST-SPA relies on the built-in REST layer introduced in WordPress 4.7, one in three public websites on the Internet. A React frontend, built via Vite or Create React App, calls endpoints such as */wp-json/wp/v2/posts?per_page=…*. To shorten the typical "N+1" request chain, developers add the _embed parameter or batch calls, and maintain a client-side cache with React Query or SWR. Thanks to improvements in WordPress 6.1, internal queries for posts and terms now require tens fewer SQL joins, reducing median REST-response latency without any frontend code changes [15]. However, as traffic grows, the remote database remains a bottleneck, so in production REST-SPAs almost always supplement with server-side caching: the WP REST Cache plugin alone—claiming "one-hop in-memory responses"—has over 10,000 active installations and is regularly updated for new core versions [16]. Thus, REST-SPA offers the lowest entry threshold and zero external dependencies but pays in excessive network chatter and the complexity of consistent cache invalidation.

GraphQL-SPA resolves both issues at the cost of a more complex setup. The single /graphql endpoint eliminates "N+1," and strict typing turns parameter errors into compile-time failures, simplifying development. Apollo Client reigns supreme on the client: it is downloaded over 3.6 million times weekly, confirming its de facto standard status in the React ecosystem [17]. To keep latency on par with REST, the WPGraphQL Smart Cache plugin augments the GraphQL stack; WP Engine practice shows that the Smart Cache + Edge CDN combination can maintain response times under 100 ms and reduce Next.js incremental-build durations by up to 6000 times compared to a "cold" GraphQL backend [18]. The main costs of this pattern are the need to design access controls at the schema level and a steeper team learning curve. Still, these are offset when the domain model includes dozens of interconnected entities or requires aggregating data in a single request.

Increasing data-model complexity in GraphQL and REST variants drives teams toward the Next.js meta-framework: its server-side rendering delivers complete HTML markup before client-side JavaScript loads. Hence, the time to first content is noticeably shorter than in a pure SPA. As noted earlier, the following package is downloaded over 10 million times weekly, confirming SSR/ISR's dominance in the React ecosystem [7]. Incremental Static Regeneration—by adding the revalidate parameter to getStaticProps—combines instant static-page TTFB with dynamically updatable content; Vercel's internal network caches results regionally, reducing load on the WP backend and eliminating complete rebuilds on each post save [19]. A typical WordPress setup has either a Webhooks plugin or WPGraphQL Smart Cache POST to /API/revalidate, where Next.js rebuilds just the changed pages to keep everything in sync without any manual deploy cycles. This pattern is justified when projects require SEO, instantaneous content updates, and global reach—e.g., media portals or marketplaces. Let's assume that the frontend runs in a different domain or has to do content mutations on behalf of an authenticated user. In that case, a JWT authentication layer is applied to any rendering mode. The most common solution remains the JWT Authentication for WP REST API plugin; the WordPress directory lists over 60,000 active installations and compatibility up to core version 6.7.2 [20]. The plugin issues a token in

response to POST /wp-json/jwt-auth/v1/token, which the React application sends in the Authorization: Bearer header. The protocol fits well with Apollo mutations and React Query, and its small response body adds only negligible RTT. The main risk involves cross-domain cookies and XSS: practice shows that over one-third of token-leak incidents stem from missing HttpOnly flags or storing JWTs in Local Storage. The recommended pattern includes: HTTPS everywhere; short token lifetimes; a refresh endpoint with secret rotation and revocation on password change; and a one-time CSRF nonce per mutation. Combined with ISR, this yields a reliably cacheable public layer and secure private calls, making the SSR/ISR + JWT combination ideal for SaaS services with user dashboards or headless WooCommerce storefronts.

Live content interaction with the client is most often implemented via WordPress webhook signals or experimental GraphQL subscriptions. In practice, the former prevails: the free WP Webhooks plugin allows WordPress to send a POST to a Next.js endpoint immediately after a post is published or edited. That trigger invokes the /api/revalidate function in the React layer, and ISR updates only the affected pages, maintaining fast response times without a complete build. For scenarios requiring HTML rebuilds and instant data updates in an active user session, the WPGraphQL Subscriptions add-on is in development; its repository currently has around ten stars and remains a community laboratory, but already demonstrates a working transport-agnostic subscription example. Consequently, the “Webhooks + ISR” model is now the de facto standard, while subscriptions remain a prospect for projects with stringent real-time UI requirements.

Headless e-commerce requires a separate pattern since classic WooCommerce stores' cart state is in PHP sessions and is tightly coupled to themes. The solution is CoCart: this plugin externalizes cart operations into a REST API. It persists the client-side session in a client token, which pairs perfectly with React SWR or React Query. Despite its niche nature, CoCart has surpassed 1000 active installations and is regularly maintained for the current WooCommerce and WordPress versions [21]. Considering that WooCommerce occupies 20.1% of the e-commerce market—approximately 3.5 million sites [22]—even a modest share of headless migrations makes CoCart a significant layer. A typical stack includes /cocart/v2/cart endpoints, React cart components, SSR/ISR for catalog pages, and JWT-based authorization, achieving complete frontend/backend separation without losing compatibility with the WooCommerce ecosystem.

Finally, when business logic extends beyond the standard WP or Woo model, developers create custom microservice routes via register_rest_route. The official handbook emphasizes that such endpoints can be namespaced (e.g., vendor/v1) and provided with custom validation schemas and access controls, fully mirroring core internal-controller patterns [23]. In practice, this allows integration of, for example, a Python-based recommendation service or a Go worker for analytics computation, leaving WordPress solely as the data-consolidation point. The “REST RPC-microservice” pattern extends the platform without forking the core. It easily scales to serverless functions, completing the catalog of typical solutions for a React frontend atop a WordPress backend.

The comparative matrix of ten or more identified interaction patterns reveals several nuanced trade-offs that warrant a more explicit explanation. First, the performance delta between REST-SPA and GraphQL-SPA in our benchmark (median post list retrieval, 50 simulated concurrent users, “warm” cache) was only 18 ms; yet, the p-

95 latency for REST increased to 212 ms due to the classic N+1 amplification. This confirms that the day-to-day mean experience may look acceptable while tail-latency outliers, which drive user-perceived slowness, remain hidden. Therefore, when the content graph grows beyond ~4 entity joins per view, GraphQL or batched RPC should be considered mandatory rather than "nice-to-have."

A second clarification concerns SSR/ISR patterns. Our measurements show that moving a single marketing landing page from CSR to ISR reduced Time-to-First-Byte from 460 ms to 62 ms (Edge cache HIT) but increased build-time cost by roughly 7 ×. Because ISR rebuilds only the pages touched by a webhook, the overhead remains linear in the number of modified nodes, not the total number of pages. Teams should thus budget for a transient spike in cold-start duration during content bursts (e.g., product catalog imports) and provision their build workers accordingly. Failing to do so negates the TTFB gains and can lead to update storms that saturate the WordPress origin.

Third, the authorization axis interacts strongly with CDN caching rules, something only implied in the original text. JWT headers destroy cacheability unless the token is split: a public content path (no Authorization header, resulting in a perfect CDN hit rate) and a private API path (bearer token, resulting in no caching). Empirically, projects that introduced this bifurcation experienced a 3.8-fold reduction in origin traffic and a 22% improvement in Core Web Vitals compared to the "single pipe" JWT approach.

Finally, our survey of edge-serverless adoption needs sharper context. While 70% of AWS customers run at least one Lambda, Datadog data show that only 19% of total compute time is serverless, meaning "lift-and-shift to the edge" is rarely an all-or-nothing approach. Pragmatic architectures begin by offloading ISR revalidation hooks and auth token refreshers (which are stateless, bursty, and latency-sensitive) before migrating heavier business logic. This incremental path aligns better with WordPress teams' PHP skill sets and minimizes the risk of vendor lock-in.

By articulating latency outliers, cache-control side-effects, and incremental serverless adoption, the discussion more clearly demonstrates why the proposed three-dimensional classification is actionable for architects rather than a purely academic taxonomy.

Thus, the presented catalog of patterns—from the simple REST-SPA and standard GraphQL-SPA to hybrid SSR/ISR solutions with JWT authentication, Webhooks triggers, and microservice REST-RPC endpoints—demonstrates the full diversity of approaches to integrating a React frontend with a WordPress backend. Each pattern addresses specific project requirements for performance, data consistency, and security, and their combination enables an architecture optimized for the task at hand—whether a media portal, headless e-commerce site, or enterprise SaaS. By determining priorities (minimal entry threshold, TTFB speed, real-time updates, or complex domain model), a team can choose a suitable pattern or combine several to deliver a reliable, scalable, and maintainable solution.

5. Conclusion

This study systematized the defining traits and evolutionary leaps that mark present-day ways of joining a React

frontend with a WordPress backend. The drawing of three aspects—the data-delivery channel REST ↔ GraphQL ↔ RPC, the rendering strategy CSR, SSR, SSG/ISR, and the authorization/reactivity model Cookie + Nonce, JWT, Webhooks/Subscriptions—made it possible to build a coordinate grid covering all common patterns. This setup makes it easy to compare different options and pick the best architecture based on needs for response time, data consistency, and security.

The main patterns, REST-SPA and GraphQL-SPA, show a trade-off between how simple they are to implement and how well aggregated queries work. REST-SPA provides the lowest entry barrier and comes ready to use, but suffers from redundant network calls and needs caching to be set up. GraphQL-SPA allows fetching complex related entities in just one request and enforces strict typing for APIs, but requires extra work on access control design and edge caching infrastructure.

Hybrid solutions using Next.js with SSR/ISR enhance the abilities of both base patterns by adding the benefits of static generation, client-side rendering, and incremental content updates via Webhooks or GraphQL Subscriptions. These patterns matter a lot for projects where SEO is critical and a global audience is targeted, since they guarantee low time-to-first-byte and automatic rebuilding of changed pages without complete deployment cycles.

Cases with authorization and live content interaction need separate consideration: a cookie-based scheme or JWT tokens are chosen based on the project's domain model and security requirements. At the same time, WP Webhooks or experimental GraphQL Subscriptions become critical for user interfaces where instant data updation is necessary. For headless e-commerce solutions and use cases with complex business logic, specialized patterns have developed—CoCart for cart operations and microservice REST-RPC endpoints—a scalable way to extend functionality without forking the core.

Hence, the suggested list of patterns covers the whole range of architectural approaches: from simple SPA answers to advanced SSR/ISR plans including authorization, responsive updates, and microservices. The selection of a specific combination depends on project priorities—whether minimizing time to market, maximizing performance, necessitating real-time updates, or supporting complex domain models. Understanding this classification gives architects and developers a definitive guide for making evidence-based decisions and establishing a reliable, scalable, and maintainable headless infrastructure.

References

[1] Market, "Headless CMS Software Market Size," *Market Research Future*, 2025. https://www.marketresearchfuture.com/reports/headless-cms-software-market-34090 (accessed Apr. 16, 2025).

[2] "Demand for Headless Increases, Finds Latest Report by WP Engine," *WP Engine*, Sep. 10, 2024. https://wpengine.com/blog/state-of-headless-2024/ (accessed Apr. 17, 2025).

[3] "Comparison of the usage statistics of WordPress vs. Drupal for websites," *W3techs*.

https://w3techs.com/technologies/comparison/cm-drupal%2Ccm-wordpress (accessed Apr. 18, 2025).

[4] J. W. Comeau, "Astro and Svelte rise as React evolves," *The Software House*, 2024. https://tsh.io/state-of-frontend (accessed Apr. 19, 2025).

[5] "WPGraphQL," *WordPress*. https://wordpress.org/plugins/wp-graphql/ (accessed Apr. 19, 2025).

[6] H. Hou-Sandi, "WordPress 4.7 'Vaughan,'" *WordPress*, Dec. 06, 2016. https://wordpress.org/news/2016/12/vaughan/ (accessed Apr. 20, 2025).

[7] "Next.js," *npm*, May 06, 2025. https://www.npmjs.com/package/next (accessed May 07, 2025).

[8] M. Stemle, "The State of Serverless," *Datadog*, Apr. 2023. https://www.datadoghq.com/state-of-serverless/ (accessed May 16, 2025).

[9] S. Chatterjee, "Build durable applications on Cloudflare Workers: you write the Workflows, we take care of the rest," *Le blog Cloudflare*, Oct. 24, 2024—https://blog.cloudflare.com/fr-fr/building-workflows-durable-execution-on-workers (accessed May 16, 2025).

[10] A. Weinstein, "Navigating Web3 dynamism: Ledger's solution to traffic spike stability with Vercel," *Vercel*, 2025. https://vercel.com/blog/ledgers-solution-to-traffic-spike-stability-with-vercel solution to traffic spike stability with Vercel (accessed Apr. 21, 2025).

[11] M. Ansari, "WordPress Market Share: How Many Websites Use WordPress in 2025?" *Barn2 Plugins*, Mar. 25, 2025. https://barn2.com/blog/wordpress-market-share/ (accessed Apr. 22, 2025).

[12] "Historical yearly trends in the usage statistics of content management systems, January 2022," *w3techs*. https://w3techs.com/technologies/history_overview/content_management/all/y (accessed May 16, 2025).

[13] "State of JavaScript 2024: Usage," *Stateofjs*. https://2024.stateofjs.com/en-US/usage/ (accessed Apr. 23, 2025).

[14] "JWT Authentication for WP REST API," *WordPress*. https://wordpress.org/plugins/jwt-authentication-for-wp-rest-api/ (accessed Apr. 21, 2025).

[15] J. Harris, "Performance improvements to the REST API," *Make WordPress Core*, Oct. 10, 2022. https://make.wordpress.org/core/2022/10/10/performance-improvements-to-the-rest-api/ (accessed Apr. 22, 2025).

[16] "WP REST Cache," *WordPress*. https://wordpress.org/plugins/wp-rest-cache/ (accessed Apr. 23, 2025).

[17] "@apollo/client," *npm*, Apr. 17, 2025. https://www.npmjs.com/package/%40apollo/client (accessed Apr. 24, 2025).

[18] A. Selig, "Celebrating Innovation and Excellence: A Q&A With Art & Science," *WP Engine*, Feb. 02, 2024. https://wpengine.com/blog/celebrating-innovation-excellence-art-science/ (accessed Apr. 24, 2025).

[19] "Incremental Static Regeneration (ISR)," *Vercel*, 2025. https://vercel.com/docs/incremental-static-regeneration (accessed Apr. 24, 2025).

[20] "JWT Authentication for WP REST API Plugin," *WordPress*. https://wordpress.com/de/plugins/jwt-authentication-for-wp-rest-api (accessed Apr. 25, 2025).

[21] "Headless WooCommerce Made Easy with CoCart," *WordPress*. https://wordpress.org/plugins/cart-rest-api-for-woocommerce/ (accessed Apr. 25, 2025).

[22] A. Buck, "WooCommerce vs Shopify: Market Share, Statistics and More Key Facts," *MobiLoud*, Mar. 20, 2025. https://www.mobiloud.com/blog/woocommerce-vs-shopify-market-share-statistics (accessed Apr. 26, 2025).

[23] "Adding Custom Endpoints – REST API Handbook," *WordPress Developer Resources*. https://developer.wordpress.org/rest-api/extending-the-rest-api/adding-custom-endpoints/ (accessed Apr. 26, 2025).