The Pedagogical Interface: Architectural Patterns for Generative UI with Next.js 15 and Vercel AI SDK

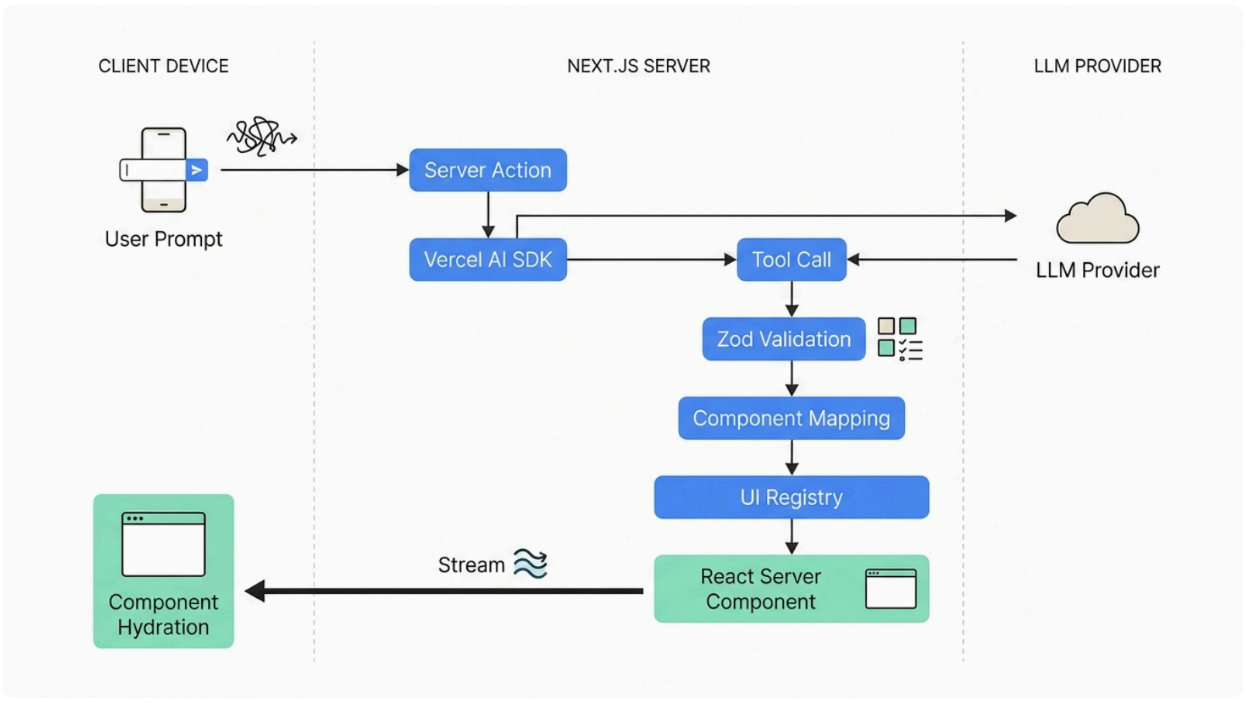
1. Executive Summary: The Evolution from Chatbots to Interface Agents

The trajectory of human-computer interaction (HCI) has historically been defined by a progressive reduction in abstraction. Command-line interfaces required users to memorize syntax; graphical user interfaces (GUIs) replaced syntax with spatial metaphors; and touch interfaces removed the abstraction of the mouse. The current epoch, dominated by Large Language Models (LLMs), initially seemed to revert to a command-based paradigm—the "chat" interface. While natural language processing allows for infinite expression, the output modality—unstructured text—remains a significant bottleneck for complex information transfer. We are now witnessing the emergence of the **Generative User Interface (GenUI)**, a paradigm where the AI does not merely describe a concept but dynamically constructs the interface required to understand it.

This report provides an exhaustive architectural analysis of implementing GenUI using **Next.js 15**, **React Server Components (RSC)**, and the **Vercel AI SDK**. We explore the transition from static, pre-defined user flows to "Context-Aware" interfaces that adapt in real-time to user intent. Specifically, we examine the "Pedagogical Level" of interaction, where an AI detects user confusion regarding complex topics—such as **FSRS (Free Spaced Repetition Scheduler) Retention Curves**—and responds not with a text explanation, but with a live, interactive **Recharts** visualization streamed directly from the server.

The core of this architecture is the **Prompt-to-Component Pipeline**: a mechanism that transmutes unstructured natural language prompts into structured, validated JSON instructions, which then trigger the rendering of safe, client-side React components. We analyze the critical role of **Server Actions** as the transport layer for these interactions, the necessity of **Zod** schema validation for preventing Cross-Site Scripting (XSS) in AI-generated props, and the complex synchronization required to bridge the gap between server-side AI logic and client-side component state.[1]

Architecture of the Prompt-to-Component Pipeline



The pipeline begins with a User Prompt triggering a Server Action. The Vercel AI SDK invokes the LLM, which outputs a Tool Call. This call is intercepted, validated against a Zod Schema, and mapped to a component in the UI Registry. The resulting React Server Component is streamed back to the client.

---

2. Theoretical Foundations: The Limitations of Text-Based Interaction

2.1 The Cognitive Bandwidth Problem

The fundamental premise of Generative UI is that text is an inefficient medium for conveying structural, relational, or quantitative information. When a user inquires about "FSRS Retention Curves," they are engaging with a mathematical concept governed by a decay function, typically expressed as $R = (1 + \text{factor} \times \frac{t}{s})^{\text{decay}}$. In a standard chat interface, the AI might output this formula or a static markdown table of values. However, to truly understand the relationship between "Stability" (the strength of the memory) and "Retention" (the probability of recall over time), the user must perform a mental simulation of the formula. This imposes a high cognitive load, effectively using the user's working memory as a rendering engine.

Generative UI shifts this burden from the user to the system. By rendering a visual curve, the system utilizes the human visual cortex's ability to process pattern and trend information instantaneously—a bandwidth difference of orders of magnitude compared to reading text. This is the "Pedagogical Level": the interface acts as a teacher that selects the optimal medium for the lesson. If the lesson involves code, it renders a sandbox; if it involves geography, a map; if it involves data trends, an interactive chart.[4]

2.2 The React Server Component (RSC) Paradigm Shift

The implementation of such dynamic interfaces has historically been hampered by the heavy client-side JavaScript required to support them. In a traditional Single Page Application (SPA), the client must download the code for every potential component the AI *might* suggest, leading to bloated bundles and slow hydration.

Next.js 15 and **React Server Components (RSC)** fundamentally alter this equation. In the RSC model, components can be rendered on the server. The server sends a serialized description of the UI (the "Flight" protocol) rather than the raw JavaScript bundle. This allows a GenUI application to possess a vast library of potential components—Charts, Maps, Quizzes, 3D Models—without penalizing the initial load time. The AI, running on the server via the Vercel AI SDK, "decides" which component to use, renders it, and streams the result. The client only hydrates the interactive "leaves" of the tree (e.g., the specific slider on the graph), keeping the application performant and responsive.[3]

2.3 The "Context-Aware" Imperative

The user query highlights the need for a "Context-aware AI chat." True context awareness extends beyond simply remembering previous messages. It implies an understanding of the *current state of the user interface*. If a user is looking at a graph and says "Flatten the curve," the AI must understand that "Flatten" implies modifying the decay or stability parameters of the currently visible component. This requires a bidirectional flow of information: the server streams UI to the client, and the client streams state updates back to the server. This closed loop is the defining characteristic of a mature GenUI architecture.[5]

---

3. The Architecture of the Prompt-to-Component Pipeline

The core technical challenge in Generative UI is establishing a robust pipeline that translates the inherent ambiguity of natural language into the strict rigidity of software code. This pipeline must be deterministic, secure, and type-safe. We define this as the **Prompt-to-Component Pipeline**.

3.1 The Entry Point: Next.js 15 Server Actions

In Next.js 15, **Server Actions** serve as the primary entry point for user interactions. Unlike traditional API routes, Server Actions are asynchronous functions that can be invoked directly from client components, behaving like remote procedure calls (RPCs). This integration simplifies the architecture by co-locating the triggering logic (the user's form submission or button click) with the execution logic (the AI processing).[6]

The pipeline begins when the user submits a prompt via a <ChatInput /> component. This triggers a Server Action, for example, submitUserMessage(formData). Next.js 15 introduces useActionState (formerly useFormState), which manages the lifecycle of this action, providing pending, success, and error states to the client. This allows the UI to immediately show a "Thinking..." indicator or an optimistic update while the server begins the heavy lifting of AI inference.[6]

3.2 The Orchestrator: Vercel AI SDK

Inside the Server Action, the **Vercel AI SDK** acts as the orchestrator. The SDK abstracts the complexity of communicating with Large Language Models (LLMs) and provides specific utilities for streaming UI. The critical function here is streamUI (or the newer streamText with tool calling patterns that effectively achieve the same result).

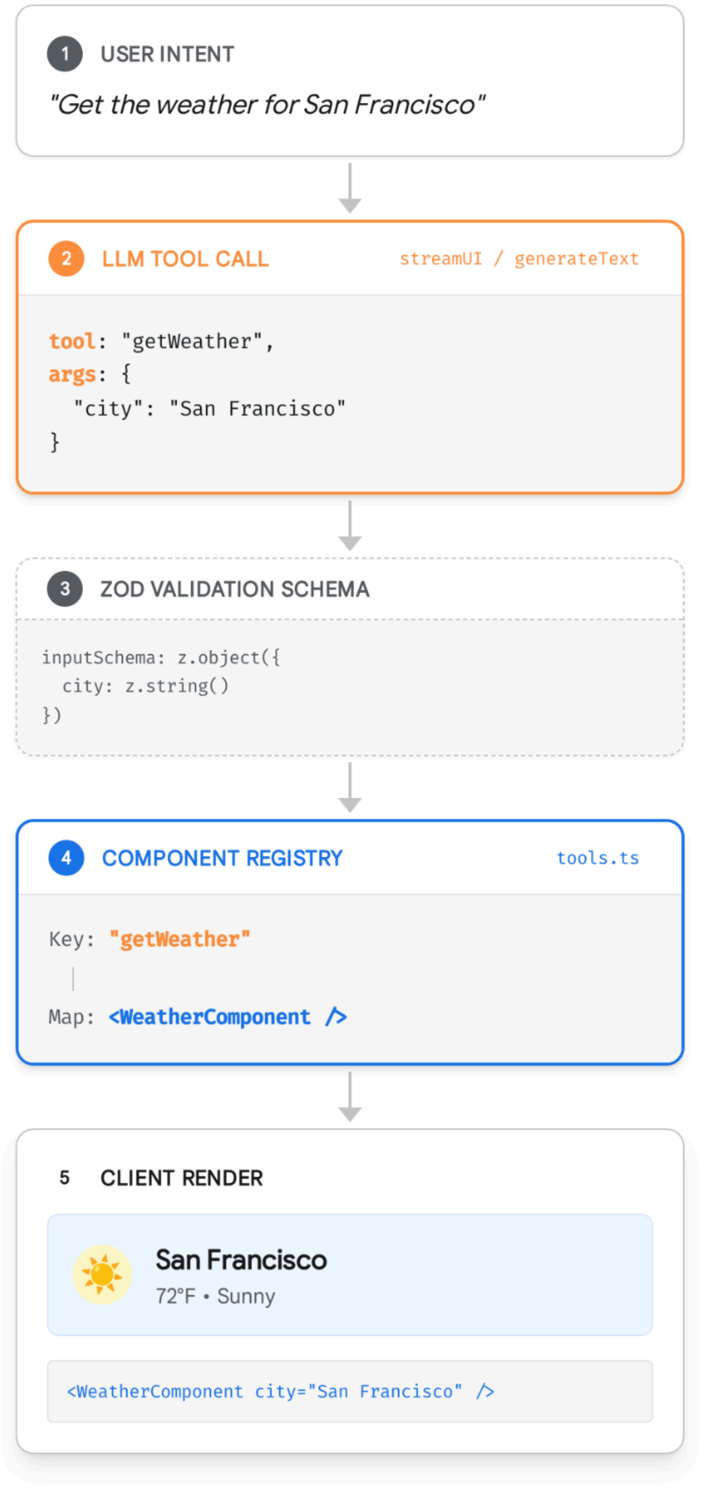
The SDK is configured with a set of "Tools." In the context of GenUI, a "Tool" is effectively a definition of a UI component that the AI is allowed to use. The AI does not "write" the component; it "calls" the tool. For instance, to render the FSRS graph, the AI calls a tool named renderRetentionGraph. It does not output React JSX code; it outputs a JSON object containing the *props* for that component.[8]

3.3 The Component Registry Pattern

The bridge between the LLM's intent and the actual React code is the **Component Registry**. This is a dictionary or a mapping object that associates a tool name with a specific React component.

The architectural significance of the Registry cannot be overstated. It acts as a strict **allow-list**. The LLM operates in a "sandbox" where it can only invoke components that have been explicitly registered. If the LLM hallucinates a tool called <SystemFileExplorer />, the Registry will fail to match it, and the system will default to a standard text response or an error state. This is a critical security feature, preventing the AI from accessing unauthorized parts of the codebase.[1]

The Registry Pattern: Mapping Intents to Components



```
tool: "getWeather",
args: {
  "city": "San Francisco"
}
```

```
inputSchema: z.object({
  city: z.string()
})
```

```
Key: "getWeather"
Map: <WeatherComponent />
```

```
<WeatherComponent city="San Francisco" />
```

The LLM selects a tool (e.g., 'display_graph'). The inputs are validated against a Zod schema (Security Layer 1). The validated props are then passed to the mapped component in the Registry (Security Layer 2).

Data sources: Vercel AI SDK RSC, Strapi React Markdown, Vercel AI SDK Tool Use

3.4 Structuring the Output: JSON and Zod

The "language" spoken between the LLM and the Registry is JSON. However, raw JSON is prone to errors—missing fields, wrong types, or hallucinated structures. To solve this, the architecture employs **Zod**, a TypeScript-first schema declaration and validation library.

Every component in the Registry must have a corresponding Zod schema. For the FSRS graph, the schema might define stability as a number between 0.1 and 100, and difficulty as a number between 1 and 10. When the AI calls the tool, the Vercel AI SDK automatically validates the output against this schema. If the validation fails (e.g., the AI outputs a string "high" instead of a number for stability), the SDK can either auto-correct, request a retry, or fail gracefully. This guarantees that the props passed to the React component are always type-safe and valid, preventing runtime crashes in the client's browser.[11]

---

4. Implementation Deep Dive: The FSRS Retention Curve

To illustrate these concepts concretely, we analyze the implementation of the requested FSRS Retention Curve visualizer. This example perfectly encapsulates the requirements: it is complex, mathematical, and requires interactivity.

4.1 The Mathematical Model and User Need

The FSRS model calculates the probability of recalling an item (R) at time (t) given its stability (S). The user's query implies a desire to intuit how changing S affects the decay of R. A static image is insufficient because the user needs to ask "What if?" questions—"What if stability is 5? What if it's 20?" The ideal interface is a line chart where the X-axis is time (days) and the Y-axis is retention probability (0-100%).

4.2 The "Hybrid" Component Strategy

A pure Server Component cannot support the requirement "user can tweak the stability inputs." Server Components are non-interactive; they run once on the server and produce HTML. To achieve interactivity, we must use a **Client Component** wrapper.

The Server Action determines the *initial* state. It calls the renderRetentionGraph tool with an initial stability value inferred from the user's prompt (e.g., user asks "How does a stability of 5 look?" -> stability: 5).

The server then streams a component tree that looks like this:

<RetentionGraphWrapper initialStability={5} />.

The RetentionGraphWrapper is a Client Component (marked with 'use client'). Upon hydration in the browser, it initializes its internal state: const = useState(initialStability).

Inside this wrapper, we utilize **Recharts** to render the line chart. Crucially, the calculation of the curve data points happens *on the client* inside a useMemo hook.

JavaScript

```javascript
const data = useMemo(() => {
 return Array.from({ length: 30 }, (_, i) => ({
   day: i,
   retention: Math.pow(0.9, i / stability) * 100
 }));
}, [stability]);
```

This architecture ensures that when the user drags the slider, the graph updates at 60 frames per second. There is no network latency, no server roundtrip, and no waiting for the AI to "regenerate" the graph. The AI provided the *tool*, but the *interaction* is native.[1]

4.3 Streaming the UI

The mechanics of getting this component to the user involve the streamUI function.

When the Server Action runs:

1. The AI determines that the user needs a graph.
2. It generates the renderRetentionGraph tool call.
3. The streamUI callback executes, instantiating the <RetentionGraphWrapper />.
4. Next.js serializes this component into the React Flight format.
5. The browser receives the stream, parses the Flight data, and mounts the component in the chat window.

This process happens progressively. The text preceding the graph streams first, followed by the graph itself, creating a smooth "typing" effect that culminates in the appearance of the rich UI.[4]

---

5. Security: Sanitizing AI-Generated Props

The prompt explicitly asks: *"How can we sanitize these AI-generated props to prevent XSS attacks?"* This is the single most critical consideration for deploying GenUI in production. AI models are trained on the public internet, including malicious code examples. A "Prompt Injection" attack could trick the AI into generating a payload designed to execute arbitrary JavaScript in the victim's browser.[13]

5.1 The Attack Surface

The primary attack vector in GenUI is **Cross-Site Scripting (XSS)** via prop injection. If the AI generates a prop such as title: "<img src=x onerror=alert(1)>", and the component renders this prop unsafely, the attacker's script executes.

While React is generally secure by default (it escapes string variables in JSX), vulnerabilities arise when:

1. Components use dangerouslySetInnerHTML.
2. Components use purely spread props <div {...props} /> where attributes like href or src can be manipulated (e.g., href="javascript:alert(1)").
3. Third-party libraries (like some charting or markdown libraries) do not sanitize inputs strictly.

5.2 Defense Layer 1: Strict Schema Validation (Zod)

The most effective defense is to prevent the malicious payload from ever reaching the rendering layer. This is done via **Zod** schemas.

We must move beyond simple type checks (e.g., "is this a string?") to semantic checks.

- **Numeric Safety:** For the stability input of the FSRS graph, we define z.number(). It is impossible to inject an XSS script into a JSON number. This renders that entire vector null and void.
- **String Constraints:** For text fields, we use regex and length limits. z.string().max(100).regex(/^[a-zA-Z0-9]*$/) ensures that no special characters (like < or >) can be passed. If the AI generates them, the validation fails, and the component is not rendered.[11]

5.3 Defense Layer 2: Sanitization Libraries

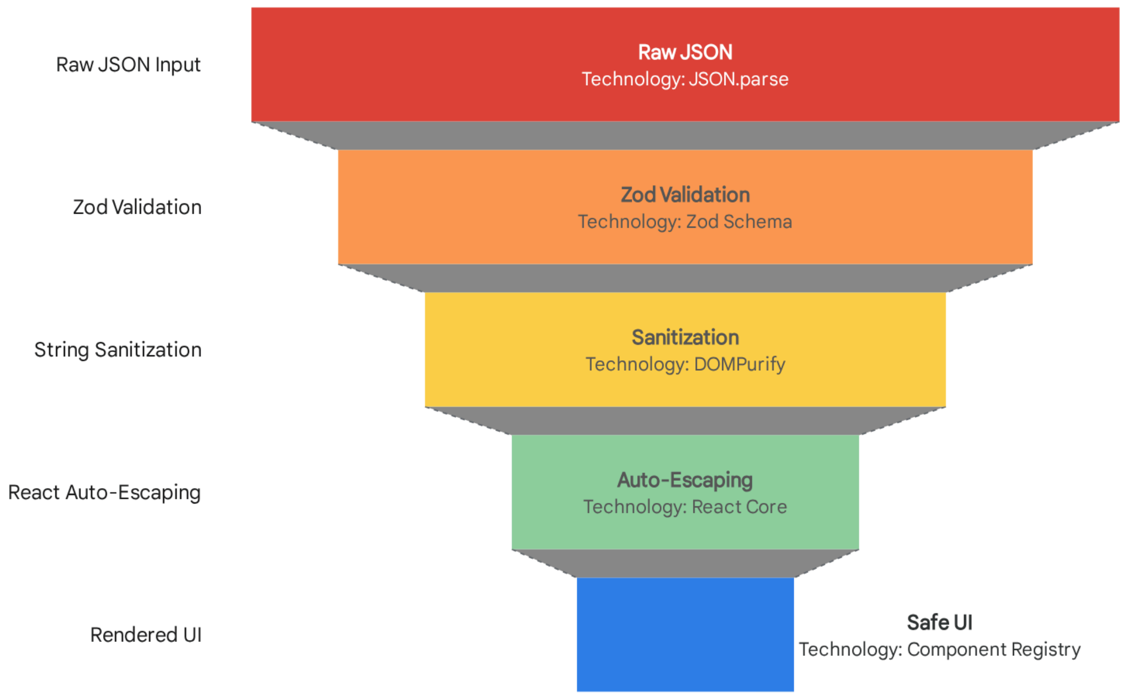
For components that *require* rich text (e.g., a "Concept Explanation" card that allows bold or italics), we cannot simply ban all HTML-like characters. Here, we must use a dedicated sanitization library like **DOMPurify**.

Ideally, sanitization should happen on the **Server** before the prop is serialized. We can use Zod's .transform() method to bake this in:

```TypeScript
const SafeHTML = z.string().transform((str) => DOMPurify.sanitize(str));
```

This ensures that the prop arriving at the client has already been scrubbed of <script>, <iframe>, and on* event handlers. This "Defense in Depth" strategy ensures that even if the Zod validation is too loose, the sanitizer catches the payload.[17]

Defense-in-Depth: The Prop Sanitization Funnel



Raw LLM output enters the top. Layer 1 (JSON Parse) ensures syntax. Layer 2 (Zod Schema) enforces strict typing, blocking script injection in numeric fields. Layer 3 (Sanitization) strips HTML tags from string fields. Only 'Clean Props' exit to the Component Registry.

Data sources: Vercel AI SDK Guide, FreeCodeCamp (Zod), DOMPurify

5.4 Defense Layer 3: Content Security Policy (CSP)

The final line of defense is the browser itself. A strict **Content Security Policy (CSP)** header prevents the execution of inline scripts and restricts the sources from which resources (images, scripts) can be loaded. Next.js 15 provides middleware support to inject nonces (random cryptographic tokens) into script tags. By configuring the CSP to only allow scripts with the correct nonce, we ensure that even if an attacker manages to inject a <script> tag into the DOM, the browser will refuse to execute it because it lacks the valid nonce.[19]

---

6. State Management: The Challenge of "Split State"

The requirement for the component to "interact with the client-side state" introduces the most complex architectural problem in GenUI: the **Split State Problem**.

- **AI State:** The LLM's memory of the conversation (Server-side).
- **UI State:** The actual value of the inputs in the browser (Client-side).

6.1 The Disconnect Scenario

Consider this user flow:

1. User: "Show me the retention curve."
2. AI: Renders graph with stability: 1.
3. User: Manually drags the slider on the graph to stability: 5.
4. User: "Why is the curve so flat?"

At step 4, the user is asking about the *current* state of the graph (stability: 5). However, the AI's conversation history only records the *initial* state (stability: 1). If the AI answers based on its history, it will hallucinate an explanation for a steep curve, confusing the user.

6.2 Solution: Client Attachments and Hidden Context

To solve this, we must synchronize the UI state back to the AI. We utilize the **Attachment** pattern supported by the Vercel AI SDK (specifically experimental_attachments or simply appending structured data to the message).

When the user types "Why is the curve so flat?", the application logic must:

1. Intercept the submission.
2. Snapshot the current state of the <RetentionGraphWrapper /> (i.e., read the stability value).
3. Attach this data to the message as a hidden "System Context" or a specific "Tool Result" payload.
4. Send the augmented message to the Server Action.

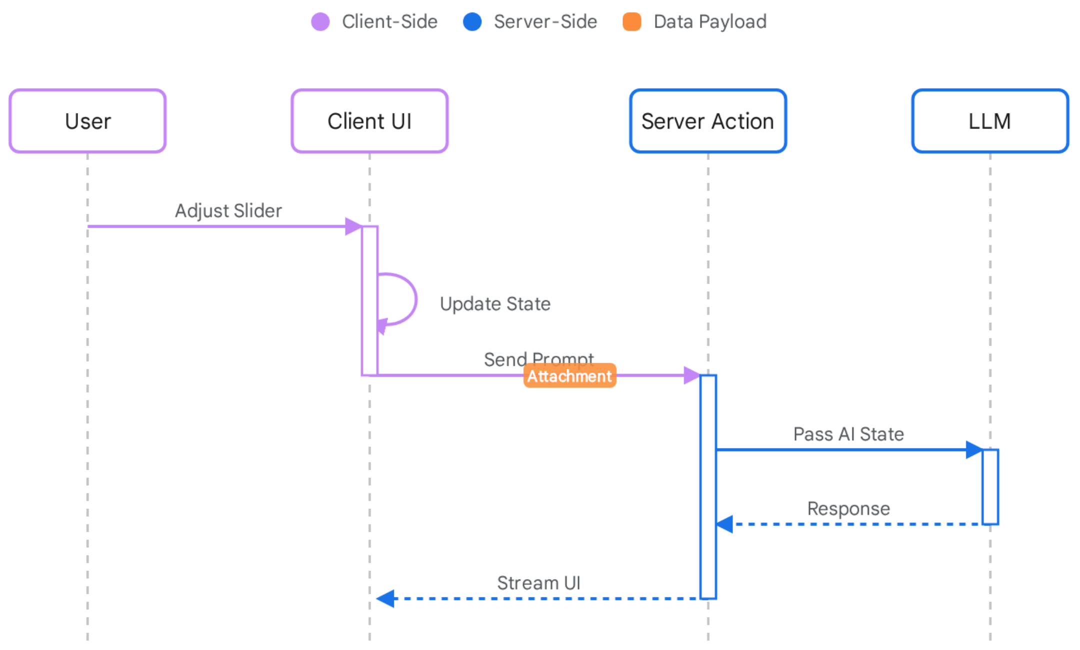
The prompt received by the LLM is effectively:

User: "Why is the curve so flat?"

System Context:

With this context, the AI can correctly analyze the parameters the user is *currently looking at*. This architectural pattern—**Client State Injection**—is essential for creating truly "Context-Aware" interfaces that feel coherent to the user.[5]

Synchronizing Client State with AI Context



1. User adjusts slider (Client State updates). 2. User sends prompt. 3. Application injects 'Client Attachment' with current slider values. 4. Server Action receives prompt + attachment. 5. LLM generates response based on the *updated* context.

Data sources: AI SDK RSC, Reddit Discussion, GelData Blog

7. Next.js 15 Implementation Specifics

The implementation of this architecture relies on specific features introduced or stabilized in

Next.js 15.

7.1 Async Request APIs

Next.js 15 has moved to asynchronous APIs for accessing request data (headers, cookies, params). In our Server Actions, this means we must await the headers to check for authentication tokens or device context before streaming the UI. This asynchronous nature aligns perfectly with the async nature of AI model calls.[3]

7.2 Caching and the AI Dynamic

Next.js is aggressive about caching. However, GenUI is inherently dynamic; the same prompt from the same user might yield different results based on the context. We must explicitly opt-out of caching for the GenUI Server Actions using export const dynamic = 'force-dynamic' or by using the no-store directive in our fetch calls to the LLM provider. This ensures that the user always gets a fresh response generated from the current state, rather than a stale cached component.[3]

7.3 Streaming Mechanics

The mechanism for streaming components from a Server Action involves the createStreamableUI (from ai/rsc) or createDataStreamResponse (from ai/core).

While ai/rsc provided the initial implementation of this pattern, the ecosystem is evolving. The robust pattern for Next.js 15 involves:

1. **Server:** Use streamText to generate the tool call.
2. **Server:** Execute the tool logic (the Zod validation and component selection).
3. **Server:** Return a StreamableUI that wraps the result.
4. **Client:** The useActions or useChat hook receives the stream.

This "Hybrid" approach leverages the stability of the core AI SDK text streaming while utilizing the RSC capabilities for the UI payload. It separates the concerns: text streams as text (fast, delta-based), and UI streams as Flight data (structured, component-based).[4]

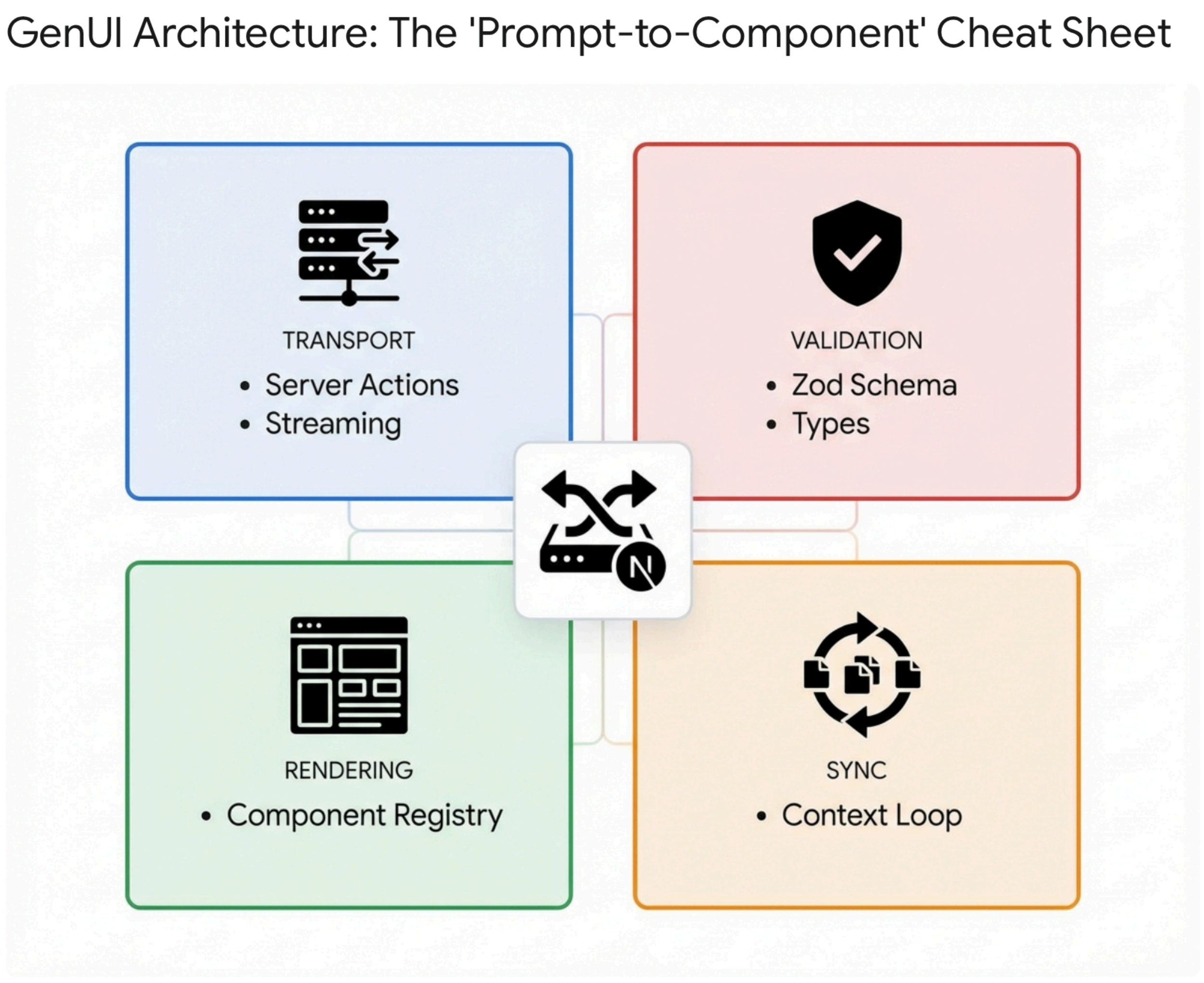
---

8. Conclusion and Future Outlook

The architectural patterns detailed in this report—**Registry-based Component Orchestration**, **Zod-guarded Security Pipelines**, and **State-Injected Context Sync**—form the blueprint for the next generation of web applications. By moving the "pedagogical" burden from text explanation to visual demonstration, we unlock a deeper level of user understanding.

The "FSRS Retention Curve" example demonstrates that the AI of 2026 is not just a writer; it is a software engineer that builds the interface the user needs, exactly when they need it. As

Next.js 15 and the Vercel AI SDK continue to mature, the friction of building these interfaces will decrease, making GenUI the standard default for all complex data interaction.



Summary of the 4 Pillars of GenUI Architecture: 1. Transport (Server Actions), 2. Validation (Zod), 3. Rendering (Registry), 4. Sync (Client Attachments).

Works cited

1. Generative UI Chatbot with React Server Components - Vercel, accessed January 22, 2026, https://vercel.com/templates/next.js/rsc-genui
2. Generative UI Patterns: Vercel AI SDK Claude Code Skill - MCP Market, accessed January 22, 2026, https://mcpmarket.com/tools/skills/generative-ui-patterns
3. Getting Started: Server and Client Components - Next.js, accessed January 22, 2026, https://nextjs.org/docs/app/getting-started/server-and-client-components

4. Introducing AI SDK 3.0 with Generative UI support - Vercel, accessed January 22, 2026, https://vercel.com/blog/ai-sdk-3-generative-ui
5. AI SDK RSC: Managing Generative UI State, accessed January 22, 2026, https://ai-sdk.dev/docs/ai-sdk-rsc/generative-ui-state
6. Next.js 15 Server Actions: Complete Guide with Real Examples (2026) | by Saad Minhas, accessed January 22, 2026, https://medium.com/@saad.minhas.codes/next-js-15-server-actions-complete-guide-with-real-examples-2026-6320fbfa01c3
7. Getting Started: Updating Data - Next.js, accessed January 22, 2026, https://nextjs.org/docs/app/getting-started/updating-data
8. Streaming React Components - AI SDK RSC, accessed January 22, 2026, https://ai-sdk.dev/docs/ai-sdk-rsc/streaming-react-components
9. AI SDK RSC: streamUI, accessed January 22, 2026, https://ai-sdk.dev/docs/reference/ai-sdk-rsc/stream-ui
10. Issues · vercel-labs/ai-sdk-preview-rsc-genui - GitHub, accessed January 22, 2026, https://github.com/vercel-labs/ai-sdk-preview-rsc-genui/issues
11. How to Use Zod for React API Validation - freeCodeCamp, accessed January 22, 2026, https://www.freecodecamp.org/news/how-to-use-zod-for-react-api-validation/
12. Mastering Zod Validation in React: A Comprehensive Guide | by Roman J. - Medium, accessed January 22, 2026, https://medium.com/@roman_j/mastering-zod-validation-in-react-a-comprehensive-guide-7c1b046547ac
13. A developer's guide to designing AI-ready frontend architecture - LogRocket Blog, accessed January 22, 2026, https://blog.logrocket.com/ai-ready-frontend-architecture-guide/
14. Understanding XSS Attacks | Vercel Knowledge Base, accessed January 22, 2026, https://vercel.com/kb/guide/understanding-xss-attacks
15. Avoiding XSS in React is Still Hard | by Ron Perris | javascript-security | Medium, accessed January 22, 2026, https://medium.com/javascript-security/avoiding-xss-in-react-is-still-hard-d2b5c7ad9412
16. How to create schema to validate react element in zod - Stack Overflow, accessed January 22, 2026, https://stackoverflow.com/questions/75663212/how-to-create-schema-to-validate-react-element-in-zod
17. How does DOMPurify ensure that sanitized HTML is safe for injection into the DOM?, accessed January 22, 2026, https://dompurify.com/how-does-dompurify-ensure-that-sanitized-html-is-safe-for-injection-into-the-dom-2/
18. Best Practices for Building Secure, Fast, and Robust React.js Applications (with Example Code) | by Qualityleaps | Medium, accessed January 22, 2026, https://medium.com/@qualityleaps/best-practices-for-building-secure-fast-and-robust-react-js-applications-with-example-code-b33533ebb157
19. Content Security Policy - Vercel, accessed January 22, 2026,

https://vercel.com/docs/headers/security-headers
20. React Security Patterns Every Developer Should Know - DEV Community, accessed January 22, 2026, https://dev.to/vanessamadison/react-security-patterns-every-developer-should-know-8ep
21. v5 - Provide React state as context to specific AI tools? : r/vercel - Reddit, accessed January 22, 2026, https://www.reddit.com/r/vercel/comments/1lsm4so/v5_provide_react_state_as_context_to_specific_ai/
22. The Next.js 15 Streaming Handbook — SSR, React Suspense, and Loading Skeleton, accessed January 22, 2026, https://www.freecodecamp.org/news/the-nextjs-15-streaming-handbook/
23. Real-time AI in Next.js: How to stream responses with the Vercel AI SDK - LogRocket Blog, accessed January 22, 2026, https://blog.logrocket.com/nextjs-vercel-ai-sdk-streaming/