Architektura Systemów Generatywnego UI (GenUI): Kompleksowa Strategia Implementacji Agenta 'Lolek' w Ekosystemie Vercel AI SDK

1. Wstęp: Paradygmatyczna Zmiana w Projektowaniu Interfejsów AI

Współczesna inżynieria oprogramowania przechodzi jedną z najbardziej fundamentalnych transformacji w swojej historii, przechodząc od statycznie definiowanych interfejsów użytkownika do systemów dynamicznych, generowanych w czasie rzeczywistym przez modele językowe. Koncepcja Generative UI (GenUI) redefiniuje rolę front-endu, przekształcając go z warstwy prezentacji sztywnych danych w elastyczne płótno, które adaptuje się do intencji użytkownika wyrażonej w języku naturalnym.[1] Dla agenta 'Lolek', którego zadaniem jest generowanie złożonych struktur interaktywnych takich jak tablice Kanban, formularze czy wykresy analityczne, wybór odpowiedniej architektury jest decyzją krytyczną, determinującą skalowalność, wydajność oraz User Experience (UX) całej aplikacji.

Analiza ekosystemu Vercel AI SDK na przełomie lat 2024 i 2025 wskazuje na znaczący zwrot technologiczny. Początkowy entuzjazm związany z renderowaniem komponentów po stronie serwera (React Server Components - RSC) przy użyciu funkcji streamUI ustąpił miejsca bardziej dojrzałemu podejściu opartemu na klienckim renderowaniu narzędzi (Client-Side Tool Rendering) przy użyciu hooka useChat.[3] Ta zmiana, podyktowana potrzebą głębszej interaktywności i lepszego zarządzania stanem, wymusza na architektach oprogramowania rewizję dotychczasowych wzorców projektowych. Niniejszy raport stanowi wyczerpujące studium techniczne, mające na celu przeprowadzenie zespołu deweloperskiego przez proces projektowania systemu komponentów dla agenta 'Lolek', z naciskiem na integrację z biblioteką Shadcn/UI, walidację schematów danych oraz implementację zaawansowanych pętli sprzężenia zwrotnego (Human-in-the-Loop).

2. Ewolucja Architektury GenUI: Od streamUI do useChat

Zrozumienie obecnych rekomendacji ("best practices") wymaga głębokiej analizy historycznej i technicznej ewolucji Vercel AI SDK. Wybór między architekturą opartą na RSC a architekturą kliencką nie jest jedynie kwestią preferencji składniowych, lecz fundamentalną decyzją wpływającą na cykl życia komponentów, zarządzanie pamięcią oraz percepcję opóźnień przez użytkownika końcowego.

2.1 Analiza Krytyczna Podejścia React Server Components (RSC) i streamUI

W początkowych fazach rozwoju Vercel AI SDK (wersje 3.x), funkcja streamUI była promowana jako rewolucyjne rozwiązanie pozwalające na strumieniowanie gotowych węzłów React bezpośrednio z serwera do klienta.[5] Model ten zakładał, że Large Language Model (LLM) decyduje o wywołaniu funkcji, która na serwerze zwraca wyrenderowany komponent (np. <WeatherCard />), a nie surowe dane JSON.

Podejście to, choć innowacyjne, napotkało na szereg barier w implementacji wysoce interaktywnych systemów agentowych:

- **Izolacja Stanu:** Komponenty strumieniowane z serwera (RSC) są z natury bezstanowe w kontekście klienta, dopóki nie zostaną "nawodnione" (hydrated) przez komponenty klienckie. Przekazywanie złożonego stanu, takiego jak pozycja kart na tablicy Kanban, między serwerem a klientem w czasie rzeczywistym okazało się trudne do synchronizacji bez skomplikowanych mechanizmów serializacji.[7]
- **Wstrzymanie Rozwoju (Deprecation Risk):** Dokumentacja techniczna oraz dyskusje w repozytorium Vercel AI SDK jednoznacznie wskazują, że rozwój pakietu @ai-sdk/rsc i funkcji streamUI został wstrzymany ("paused").[3] Jest to sygnał ostrzegawczy dla nowych projektów ("greenfield"), sugerujący, że oparcie architektury agenta 'Lolek' na streamUI wiąże się z ryzykiem długu technologicznego.
- **Ograniczenia Interakcji Zwrotnych:** W modelu streamUI obsługa zdarzeń takich jak "zatwierdzenie formularza" wymagała stosowania Server Actions i ręcznej aktualizacji stanu UI (useUIState), co komplikowało przepływ danych i utrudniało optymistyczne aktualizacje interfejsu (Optimistic UI updates).[7]

2.2 Nowy Standard: Client-Side Tool Rendering (useChat)

Obecny konsensus architektoniczny (late 2025) przesuwa odpowiedzialność za renderowanie na klienta, pozostawiając serwerowi rolę orkiestratora intencji i danych. W tym modelu, wykorzystującym hook useChat z pakietu @ai-sdk/react, LLM generuje ustrukturyzowane wywołania narzędzi (toolInvocations), które są przesyłane do klienta jako obiekty JSON.[10]

Poniższa tabela przedstawia szczegółowe porównanie implikacji obu architektur dla systemu 'Lolek':

Cecha Architektoniczna	Legacy (streamUI / RSC)	Modern (useChat / Client Tools)	Implikacja dla Agenta 'Lolek'
Przesył Danych	Serializowane węzły React (HTML-like)	Ustrukturyzowany JSON (Schema-based)	JSON pozwala na precyzyjną walidację (Zod) przed renderowaniem.
Zarządzanie Stanem	useUIState (złożona synchronizacja)	useState / useReducer (standard React)	Łatwiejsza implementacja Drag-and-Drop w Kanbanie.
Interaktywność	Wymaga hydracji i Server Actions	Natywna obsługa zdarzeń przeglądarki	Płynniejsze interakcje w formularzach i wykresach.
Typowanie (TypeScript)	Luźne powiązanie propsów	Ścisłe typowanie przez Zod Schemas	Redukcja błędów "runtime" przy generowaniu propsów.
Status Rozwoju	Wstrzymany (Paused/Experimental)	Aktywny, Rekomendowany (Production)	Stabilność długoterminowa projektu.

Modele Mentalne	Serwer "rysuje" interfejs	Serwer "deklaruje" intencję, Klient "rysuje"	Lepsza separacja logiki biznesowej od prezentacyjnej.

2.3 Hybrydowa Strategia Migracji

Dla agenta 'Lolek' rekomendowana jest pełna adopcja modelu **Client-Side Tool Rendering**. Oznacza to, że agent nie będzie "zwracał komponentu React", lecz "wywoływał narzędzie", którego wynikiem jest renderowanie komponentu w strumieniu czatu po stronie klienta. Podejście to, wspierane przez Vercel AI SDK 3.3+, pozwala na wykorzystanie pełnej mocy bibliotek takich jak Shadcn/UI, które są z natury komponentami klienckimi lub hybrydowymi.[12]

3. Projektowanie Systemu Komponentów (Rejestr 'Lolek UI')

Kluczowym elementem sukcesu w implementacji GenUI jest stworzenie robustnego systemu mapowania intencji AI na konkretne wizualizacje. Nie możemy pozwolić modelowi na swobodne generowanie kodu JSX (co jest podatne na błędy i ataki XSS); zamiast tego tworzymy deterministyczny rejestr komponentów.

3.1 Wzorzec Rejestru Narzędzi (Tool Invocation Switch)

Sercem warstwy prezentacyjnej agenta 'Lolek' jest mechanizm renderowania warunkowego oparty na analizie strumienia wiadomości. W hooku useChat, każda wiadomość asystenta może zawierać tablicę parts, w której znajdują się segmenty tekstu oraz wywołania narzędzi (tool-invocation).[14]

Architektura tego rozwiązania opiera się na komponencie "Routera", który iteruje po częściach wiadomości i deleguje renderowanie do wyspecjalizowanych widoków.

TypeScript

```
// Konceptualny model Routera Komponentów dla Loleka
{message.parts.map((part, index) => {
 if (part.type === 'tool-invocation') {
  const { toolName, toolInvocation } = part;

  // Switch-case sterujący wyborem komponentu na podstawie nazwy narzędzia
  switch (toolName) {
   case 'generateKanbanBoard':
    return (
     <KanbanView
        key={toolInvocation.toolCallId}
        toolInvocation={toolInvocation}
        isInteractive={true}
      />
    );
   case 'renderAnalyticsChart':
    return (
     <ChartView
        key={toolInvocation.toolCallId}
        toolInvocation={toolInvocation}
      />
    );
   case 'dynamicFormWizard':
    return (
     <FormWizard
        key={toolInvocation.toolCallId}
```

```
          toolInvocation={toolInvocation}
        />
      );
    default:
      // Fallback dla nieznanych narzędzi lub ładowania
      return <ToolFallback toolName={toolName} />;
  }
 }
 return <TextPart key={index} content={part.text} />;
})}
```

3.2 Integracja z Shadcn/UI i "AI Elements"

Biblioteka Shadcn/UI stanowi idealny fundament dla GenUI ze względu na swoją architekturę "headless" (niezależność od stylów) oraz dostępność kodu źródłowego, co pozwala na głęboką kastomizację.[12] Vercel wprowadził również koncepcję "AI Elements" – gotowych komponentów zoptymalizowanych pod kątem interakcji z AI, które warto zaadaptować w systemie Loleka.[17]

Strategia integracji Shadcn/UI w systemie Lolek powinna opierać się na tworzeniu **Komponentów Agentowych (Agent Components)**. Są to wysokopoziomowe wrappery, które:

1. Przyjmują surowe dane z wywołania narzędzia (zgodne ze schematem Zod).
2. Zarządzają wewnętrznym stanem interfejsu (np. otwarte/zamknięte akordeony, aktywna zakładka).
3. Komponują atomy Shadcn (Button, Card, Input) w spójną całość.

Na przykład, zamiast pozwalać AI na generowanie dziesięciu oddzielnych przycisków, tworzymy komponent <ActionGroup actions={['save', 'delete']} />, który wewnętrznie renderuje odpowiednie komponenty Button z Shadcn, dbając o spójność wizualną i dostępność (accessibility).

3.3 Obsługa Stanów Przejściowych (Generative States)

W architekturze opartej na useChat, każde wywołanie narzędzia posiada stan (state), który determinuje sposób renderowania komponentu. System Loleka musi obsługiwać trzy kluczowe fazy cyklu życia komponentu generatywnego [4]:

1. **input-streaming / call**: Model wciąż generuje argumenty dla narzędzia.
    - *Rekomendacja UX:* Wyświetlanie "Szkieletu" (Skeleton Loader) lub wskaźnika "Lolek projektuje tablicę...". Dla prostych danych (tekst) można pokazywać strumieniowany podgląd w czasie rzeczywistym.
2. **result / output-available**: Narzędzie zostało wykonane (na serwerze lub kliencie) i wynik jest dostępny.
    - *Rekomendacja UX:* Renderowanie pełnego, interaktywnego komponentu (np. gotowej tablicy Kanban). To jest moment, w którym użytkownik przejmuje kontrolę.
3. **error**: Wystąpił błąd podczas generowania lub wykonywania.
    - *Rekomendacja UX:* Wyświetlenie komunikatu błędu z możliwością ponowienia ("Retry"). Dzięki addToolResult (omówionemu w sekcji 5) możliwe jest przesłanie informacji o błędzie z powrotem do modelu, aby ten spróbował auto-korekty.[15]

4. Studium Przypadku: Implementacja Tablicy Kanban

Tablica Kanban stanowi jeden z najbardziej złożonych elementów GenUI ze względu na zagnieżdżoną strukturę danych i wysoki poziom interaktywności (Drag-and-Drop). Analiza tego przypadku pozwala zilustrować kluczowe wyzwania inżynieryjne.

4.1 Rekurencyjne Schematy Zod dla Struktur Złożonych

Definicja narzędzia (Tool Definition) wymaga precyzyjnego schematu walidacji. W przypadku Kanbana, struktura jest hierarchiczna (Tablica -> Kolumny -> Karty). Wykorzystanie biblioteki Zod jest tutaj standardem branżowym, zapewniającym bezpieczeństwo typów od momentu wygenerowania JSON przez LLM aż do renderowania w React.[19]

TypeScript

```
// schemas/kanban.ts
import { z } from 'zod';

// Definicja pojedynczego zadania
const taskSchema = z.object({
  id: z.string().describe("Unikalny identyfikator UUID dla operacji Drag-and-Drop"),
  content: z.string().describe("Treść zadania"),
  priority: z.enum(['low', 'medium', 'high']).optional(),
  tags: z.array(z.string()).optional()
});

// Definicja kolumny
const columnSchema = z.object({
  id: z.string(),
  title: z.string().describe("Nazwa kolumny, np. 'Do Zrobienia', 'W Trakcie'"),
  tasks: z.array(taskSchema)
});

// Główny schemat narzędzia
export const kanbanBoardSchema = z.object({
  title: z.string().describe("Tytuł projektu"),
  columns: z.array(columnSchema).min(2).describe("Musi zawierać przynajmniej kolumny startową i końcową")
});
```

Insight Ekspercki: Użycie metody .describe() w Zod jest kluczowe. Działa ona jak "mikro-prompt" dla modelu, instruując go np. o konieczności generowania unikalnych ID, co jest niezbędne dla poprawnego działania bibliotek Drag-and-Drop (np. dnd-kit lub react-beautiful-dnd).[20]

4.2 Synchronizacja Stanu i Optimistic UI

Kiedy Lolek wygeneruje tablicę, dane trafiają do komponentu jako props z toolInvocation. Jednakże, tablica Kanban musi być interaktywna. Użytkownik przesuwa kartę z kolumny A do B.

Problem: Dane pochodzące z useChat (strumień wiadomości) są w pewnym sensie "niezmienne" (immutable history).
Rozwiązanie: Komponent <KanbanView> musi inicjalizować swój lokalny stan na podstawie propsów.

TypeScript

```
// components/ai-elements/kanban-view.tsx
export function KanbanView({ toolInvocation, addToolResult }: KanbanProps) {
 // Inicjalizacja stanu lokalnego z danych wygenerowanych przez AI
 const = useState(toolInvocation.args);

 // Obsługa Drag-and-Drop
 const onDragEnd = (result) => {
  // 1. Aktualizacja optymistyczna (lokalna) - natychmiastowa reakcja UI
  const newData = reorderBoard(boardData, result);
  setBoardData(newData);

  // 2. (Opcjonalnie) Synchronizacja z "mózgiem" agenta
  // Jeśli zmiana jest znacząca, możemy wysłać sygnał do LLM
 };

 return (
  <DragDropContext onDragEnd={onDragEnd}>
     {/* Renderowanie kolumn i kart przy użyciu komponentów Shadcn Card */}
```

```
    </DragDropContext>
  );
}
```

Analiza wskazuje na problem "migania" (flashing) interfejsu przy próbie synchronizacji stanu lokalnego z historią czatu.[22] Aby tego uniknąć, należy traktować wygenerowany UI jako "szkic" (draft). Dopiero wyraźna akcja użytkownika (np. przycisk "Zapisz Tablicę") powinna triggerować aktualizację globalnego stanu aplikacji lub bazy danych poprzez addToolResult lub oddzielną Server Action.

5. Interakcje Zwrotne: Pętla addToolResult i Human-in-the-Loop

GenUI to nie tylko wyświetlanie, to dialog. System Loleka musi umożliwiać użytkownikowi korygowanie wygenerowanych treści. Vercel AI SDK realizuje to poprzez mechanizm addToolResult, który pozwala na przesłanie wyniku działania narzędzia (w tym wyniku interakcji użytkownika) z powrotem do kontekstu konwersacji.[19]

5.1 Mechanizm Zatwierdzania i Korekty

Załóżmy scenariusz, w którym Lolek generuje formularz rekrutacyjny, ale użytkownik chce dodać pole "Oczekiwania finansowe".

1. **Generacja:** Model wywołuje narzędzie generateForm.
2. **Renderowanie:** Wyświetla się komponent formularza z przyciskiem "Edytuj" lub "Zatwierdź".
3. **Interakcja:** Użytkownik klika "Zatwierdź".
4. **Sprzężenie Zwrotne:** Komponent wywołuje funkcję addToolResult:
   ```
   TypeScript
   addToolResult({
     toolCallId: toolInvocation.toolCallId,
     result: { status: 'approved', formData: currentData }
   });
   ```
5. **Reakcja Modelu:** Wysłanie wyniku narzędzia automatycznie (lub konfigurowalnie)

wyzwala nową generację (trigger new generation). Model otrzymuje informację, że formularz został zatwierdzony i może przejść do kolejnego kroku, np. "Dziękuję, zapisałem formularz. Co robimy dalej?".[24]

5.2 Strategia "Human-in-the-Loop"

W systemach krytycznych (np. finansowych, medycznych), AI nie może podejmować ostatecznych decyzji. Architektura Loleka powinna wymuszać krok "Approval" w definicji narzędzia.

- Narzędzie executeTransfer nie wykonuje przelewu od razu.
- Zwraca ono stan requires_confirmation do klienta.
- UI wyświetla przycisk "Potwierdź Przelew".
- Dopiero kliknięcie przycisku wysyła addToolResult, który zawiera kryptograficzny token potwierdzenia lub po prostu flagę confirmed: true.
- Model (serwer) odbiera ten wynik i dopiero wtedy finalizuje transakcję w kolejnym kroku (Multi-step reasoning).[25]

6. Komponenty Danych: Wykresy i Formularze

Poza Kanbanem, Lolek ma generować wykresy i formularze. Tutaj kluczowe jest rozdzielenie "konfiguracji" od "danych".

6.1 Wykresy (Charts) z Recharts

Generowanie wykresów przez LLM jest podatne na halucynacje danych. Dlatego zaleca się strategię, w której model generuje **konfigurację widoku** i **zapytanie o dane**, a nie same punkty danych (chyba że zbiór jest mały).

- **Wrapper Komponentu:** Należy stworzyć komponent <ChartWrapper type="bar" config={...} data={...} />, który wykorzystuje bibliotekę Recharts (standard w ekosystemie React/Next.js).[19]
- **Design System:** Kolory wykresów powinny być narzucone przez system (np. paleta CSS variables z Shadcn), a nie generowane przez model jako kody HEX, co zapewnia

spójność wizualną (Dark/Light mode).

- **Interaktywność:** Wykresy powinny obsługiwać Tooltip i Legend z Recharts, co jest funkcjonalnością "darmową" przy użyciu tej biblioteki, a znacznie podnosi UX w porównaniu do statycznych obrazów SVG.

6.2 Dynamiczne Formularze

Dla formularzy rekomenduje się podejście "Schema-Driven UI".

1. Model generuje schemat JSON opisujący pola (typ, etykieta, walidacja).
2. Klient wykorzystuje react-hook-form oraz zod-resolver.
3. **Dynamiczna Walidacja:** Kluczowym elementem jest dynamiczne generowanie schematu Zod po stronie klienta na podstawie JSON-a otrzymanego od modelu. Pozwala to na natychmiastową walidację (np. "Email jest niepoprawny") bez konieczności pytania modelu o poprawność każdego znaku.[21]

7. Warstwa Serwerowa i Definicje Narzędzi

Choć ciężar renderowania spoczywa na kliencie, inteligencja rezyduje na serwerze w pliku route.ts.

7.1 Konfiguracja streamText

Wykorzystanie streamText z Vercel AI SDK Core jest fundamentem backendu. Należy zwrócić uwagę na parametr maxSteps.

TypeScript

```
// app/api/chat/route.ts
export async function POST(req: Request) {
```

```
  const { messages } = await req.json();

  const result = streamText({
    model: openai('gpt-4-turbo'),
    messages,
    tools: {
      // Definicja narzędzia generującego Kanban
      generate_kanban: tool({
        description: 'Tworzy nową tablicę Kanban',
        parameters: kanbanBoardSchema, // Importowany schemat Zod
        execute: async (args) => {
          // Tutaj możemy zapisać wstępny stan w bazie danych
          // lub po prostu zwrócić argumenty do klienta
          return {...args, status: 'generated' };
        },
      }),
    },
    maxSteps: 5, // Kluczowe dla pętli agentowych (Tool Chaining)
  });

  return result.toDataStreamResponse();
}
```

Ustawienie maxSteps na wartość wyższą niż 1 (np. 5) pozwala agentowi na wykonywanie sekwencji działań: "Pobierz dane pogodowe" -> "Wygeneruj wykres na ich podstawie" w jednej interakcji użytkownika.[15]

8. Migracja z streamUI i Strategia Wdrożenia

Dla zespołów, które rozpoczęły prace z streamUI, migracja do useChat jest koniecznością w obliczu wstrzymania rozwoju RSC SDK.

Kroki Migracji [4]:

1. **Backend:** Zastąpienie Server Actions (createAI, streamUI) przez Route Handler (streamText). Narzędzia przestają zwracać komponenty JSX, a zaczynają zwracać obiekty danych.
2. **Frontend:** Usunięcie useUIState i useActions. Zastąpienie ich hookiem useChat.
3. **Registry:** Przeniesienie logiki renderowania z definicji narzędzi (na serwerze) do komponentu "Switch" (na kliencie).
4. **Loading States:** Zastąpienie parametru loadingComponent z streamUI logiką sprawdzania part.state === 'input-streaming' po stronie klienta.[4]

9. Podsumowanie i Przyszłość: MCP i Agenci Autonomiczni

Projektując system dla agenta 'Lolek', należy patrzeć w przyszłość. Vercel AI SDK ewoluuje w kierunku wsparcia dla Model Context Protocol (MCP).[13] Oparcie architektury na ścisłych schematach Zod i separacja logiki narzędzi od UI pozycjonuje system Loleka w doskonałym miejscu do przyszłej integracji z ekosystemem MCP. Pozwoli to w przyszłości na udostępnianie funkcjonalności Loleka innym agentom lub platformom w sposób ustandaryzowany.

Rekomendowana architektura **Client-Side Tool Rendering** zapewnia optymalny balans między kontrolą nad UI (Shadcn), bezpieczeństwem danych (Zod) a elastycznością interakcji (React State). Jest to rozwiązanie stabilne, skalowalne i gotowe na wyzwania nowoczesnych aplikacji AI.

Works cited

1. Generative User Interfaces - AI SDK UI, accessed November 25, 2025, https://ai-sdk.dev/docs/ai-sdk-ui/generative-user-interfaces
2. Generative UI Guide 2025: 15 Best Practices & Examples - Mockplus, accessed November 25, 2025, https://www.mockplus.com/blog/post/gui-guide
3. Migrating from AI SDK RSC to UI · Issue #368 · miurla/morphic · GitHub, accessed November 25, 2025, https://github.com/miurla/morphic/issues/368
4. Migrating from RSC to UI - AI SDK, accessed November 25, 2025, https://ai-sdk.dev/docs/ai-sdk-rsc/migrating-to-ui
5. Generative UI Chatbot with React Server Components - Vercel, accessed November 25, 2025, https://vercel.com/templates/next.js/rsc-genui
6. Introducing AI SDK 3.0 with Generative UI support - Vercel, accessed November 25, 2025, https://vercel.com/blog/ai-sdk-3-generative-ui
7. AI SDK RSC: Managing Generative UI State, accessed November 25, 2025, https://ai-sdk.dev/docs/ai-sdk-rsc/generative-ui-state
8. Generative UI Chatbot with React Server Components - Vercel, accessed

November 25, 2025, https://vercel.com/new/atec-ve/templates/ai/rsc-genui
9. AI SDK RSC: Migrating from RSC to UI, accessed November 25, 2025, https://sdk.vercel.ai/docs/ai-sdk-rsc/migrating-to-ui
10. Advanced: Rendering UI with Language Models - AI SDK, accessed November 25, 2025, https://ai-sdk.dev/docs/advanced/rendering-ui-with-language-models
11. useChat - AI SDK UI, accessed November 25, 2025, https://ai-sdk.dev/docs/reference/ai-sdk-ui/use-chat
12. The Foundation for your Design System - shadcn/ui, accessed November 25, 2025, https://ui.shadcn.com/
13. AI SDK by Vercel, accessed November 25, 2025, https://ai-sdk.dev/docs/introduction
14. Migrating to Message Parts | Chat SDK by Vercel, accessed November 25, 2025, https://chat-sdk.dev/docs/migration-guides/message-parts
15. AI SDK 5 - Vercel, accessed November 25, 2025, https://vercel.com/blog/ai-sdk-5
16. Vercel Releases AI Elements Library for React UI Integration - InfoQ, accessed November 25, 2025, https://www.infoq.com/news/2025/08/vercel-ai-sdk/
17. Introducing AI Elements: Prebuilt, composable AI SDK components - Vercel, accessed November 25, 2025, https://vercel.com/changelog/introducing-ai-elements
18. Introduction - AI SDK, accessed November 25, 2025, https://ai-sdk.dev/elements
19. AI SDK - Vercel, accessed November 25, 2025, https://vercel.com/docs/ai-sdk
20. AI SDK Core: zodSchema, accessed November 25, 2025, https://ai-sdk.dev/docs/reference/ai-sdk-core/zod-schema
21. Structured Data Extraction | Vercel Academy, accessed November 25, 2025, https://vercel.com/academy/ai-sdk/structured-data-extraction
22. addToolResult and append causes weird flashes - AI SDK - Vercel Community, accessed November 25, 2025, https://community.vercel.com/t/addtoolresult-and-append-causes-weird-flashes/9090
23. Introducing Vercel AI SDK 3.2, accessed November 25, 2025, https://vercel.com/blog/introducing-vercel-ai-sdk-3-2
24. How to build AI Agents with Vercel and the AI SDK, accessed November 25, 2025, https://vercel.com/guides/how-to-build-ai-agents-with-vercel-and-the-ai-sdk
25. Human-in-the-Loop Agent with Next.js - AI SDK, accessed November 25, 2025, https://ai-sdk.dev/cookbook/next/human-in-the-loop
26. Multi-Step & Generative UI | Vercel Academy, accessed November 25, 2025, https://vercel.com/academy/ai-sdk/multi-step-and-generative-ui
27. Tool Calling With Vercel's AI SDK - AI Hero, accessed November 25, 2025, https://www.aihero.dev/tool-calls-with-vercel-ai-sdk
28. AI SDK 4.2 - Vercel, accessed November 25, 2025, https://vercel.com/blog/ai-sdk-4-2