# Project 1: The Natural Language DeFi Intent Interface – A Comprehensive Technical Report

## Executive Summary

The intersection of generative artificial intelligence and decentralized finance (DeFi) represents a fundamental shift in human-computer interaction, moving from imperative command-line or graphical interfaces to **intent-centric architectures**. "Project 1: The Natural Language DeFi Intent Interface" serves as a foundational blueprint for this transition. The objective is to construct a system where users express financial desires in natural language—"Swap 100 USDC for ETH on Base"—and the system autonomously orchestrates the necessary cryptographic transactions.

This report provides an exhaustive technical analysis and implementation guide for building this interface using a methodology known as **Vibe Coding**. Vibe Coding prioritizes flow-state engineering, leveraging AI-assisted scaffolding to bypass boilerplate and focus on high-level architectural intent. The proposed architecture utilizes the **Next.js App Router** for full-stack infrastructure, the **Vercel AI SDK** for streaming intelligence, and **Wagmi v2** for robust blockchain state management.

Key to this implementation is the resolution of the inherent conflict between server-side AI generation and client-side blockchain interactions. This report details the **Client-Side Generative UI pattern**, a solution that streams structured intent data (tool calls) from the server to the client, where it is hydrated into interactive, secure React components. This approach mitigates the performance bottlenecks and state synchronization issues observed in earlier React Server Component (RSC) experiments.1 Furthermore, the report establishes a rigorous protocol for **Server-Side Rendering (SSR) persistence** using cookie-based storage to prevent hydration mismatches common in Web3 applications.2

## 1. The Paradigm Shift: From Transactions to Intents

### 1.1 The Evolution of DeFi Interfaces

Traditional DeFi interfaces are deterministic and imperative. A user engaging with a protocol like Uniswap must intimately understand the mechanics: select a token, approve spending caps, select an output token, adjust slippage, and sign the transaction. This complexity acts as a significant barrier to entry.

The **Intent-Centric Interface** abstracts these mechanics. The user provides the "what" (the intent), and the AI "solver" determines the "how" (the execution path). In this context, the interface is no longer a static form but a dynamic conversation. The AI does not execute the transaction; rather, it *prepares* the transaction, acting as a translator between natural language and the rigid ABI (Application Binary Interface) requirements of smart contracts.

### 1.2 The "Vibe Coding" Methodology

Building such an interface requires a new development philosophy. "Vibe Coding" is an AI-native workflow where the developer functions as the architect and the Large Language Model (LLM) functions as the implementation engine.

| **Aspect** | **Traditional Coding** | **Vibe Coding** |
| --- | --- | --- |
| **Primary Output** | Syntax & Logic | Prompts & Architecture |
| **Iteration Speed** | Linear (Write -> Test -> Debug) | Exponential (Prompt -> Generate -> Refine) |
| **Focus** | Boilerplate & Implementation | Intent & User Experience |
| **Tooling** | IDE, Documentation | Cursor, v0, Copilot, AI SDK |

In this project, Vibe Coding allows us to scaffold complex Web3 integrations—which typically require hundreds of lines of ABI setup and hook configuration—in minutes. The focus shifts to defining the **Zod schemas** that constrain the AI's behavior and the **React components** that render the AI's output.3

## 2. Architectural Strategy and Stack Selection

The architecture must satisfy three competing constraints: **real-time responsiveness** (streaming AI), **blockchain security** (client-side signing), and **SEO/Performance** (Server-Side Rendering).

### 2.1 The Framework: Next.js App Router

We select the Next.js App Router for its robust handling of streaming responses and server actions. The App Router allows us to colocate the AI logic (Route Handlers) with the UI, ensuring type safety across the network boundary.

* **Why not Pages Router?** The App Router's support for React Server Components (RSC) is crucial for securely managing API keys and reducing the client-side JavaScript bundle, although we will selectively opt out of RSC for the wallet connection components.3

### 2.2 The Intelligence Layer: Vercel AI SDK Core vs. RSC

A critical architectural decision involves choosing between the experimental **RSC Generative UI** (@ai-sdk/rsc) and the stable **UI Hooks** (@ai-sdk/react).

* **The RSC Approach:** Streams actual React components from the server. Early research indicates this leads to significant issues with component remounting, flickering, and difficult state management when combined with client-side libraries like Wagmi.1
* **The Client-Side Approach (Selected):** Streams *data* (text and tool calls). The client listens to this stream and maps the tool calls to local components. This separates concerns: the server handles intelligence, and the client handles rendering and interactivity. This aligns with the "Generative UI" pattern while maintaining the stability required for financial applications.4

### 2.3 The Blockchain Layer: Wagmi v2 and Viem

Wagmi v2 provides the hooks for wallet connection and transaction management. It is built on Viem, a lightweight alternative to Ethers.js.

* **The Hydration Challenge:** A pervasive issue in Next.js Web3 apps is the "hydration mismatch." The server renders the page assuming no wallet is connected (because it has no access to localStorage), but the client hydrates with a connected wallet. This causes a layout shift or a "flash of disconnected content".2
* **The Cookie Solution:** We utilize a cookie-based storage adapter. By syncing the wallet connection state to a cookie, the Next.js server can read the user's status during the initial render pass, ensuring the HTML sent to the browser matches the client's state perfectly.2

## 3. Environment Setup and Foundation

This section details the initialization of the project, focusing on the configuration required to bridge the AI and Web3 worlds.

### 3.1 Project Scaffolding

The Vibe Coding workflow begins with a comprehensive initialization prompt. This prompt is designed to be fed into an AI-assisted IDE like Cursor to generate the correct file structure and dependency tree.

**Vibe Prompt 1: Project Initialization**

"I am acting as a Lead Frontend Engineer. Initialize a new Next.js 14 App Router project named defi-intent-interface. Configure the project with TypeScript, Tailwind CSS, and ESLint.

Immediately install the following core dependencies:

1. **AI Layer:** ai, @ai-sdk/openai, @ai-sdk/react, zod.
2. **Web3 Layer:** wagmi, viem, @tanstack/react-query.
3. **UI Utility:** clsx, tailwind-merge, lucide-react.

Create a lib/utils.ts file exporting a standard cn utility for class merging. Set up a .env.local file template requiring OPENAI\_API\_KEY, NEXT\_PUBLIC\_WALLETCONNECT\_PROJECT\_ID, and NEXT\_PUBLIC\_ALCHEMY\_KEY."

This prompts the AI to set up the environment, ensuring all disparate libraries are present. The use of zod is non-negotiable, as it underpins the structured data generation for our intent system.3

### 3.2 Component Library Integration (Shadcn/UI)

For the visual layer, we leverage Shadcn/UI. Unlike monolithic component libraries, Shadcn provides copy-pasteable components that we can modify. This is essential for Generative UI, where we might need to strip down a "Card" component to be lightweight enough for a chat stream.

**Vibe Prompt 2: Component Installation**

"Initialize Shadcn/UI in the project by running npx shadcn@latest init. Use the default settings (Slate color, CSS variables).

Once initialized, install the following components via the CLI: card, button, input, skeleton, toast, badge, avatar, and dialog. I will be using these to construct the interactive tools that the AI will generate.".7

### 3.3 The Web3 Configuration (SSR & Cookies)

This is the most technically sensitive part of the setup. We must configure Wagmi to use cookies for storage to support Next.js SSR.

Technical Implementation:

We define a config.ts that exports a getConfig function. This function uses cookieStorage to ensure that connection state persists across the server-client boundary.

TypeScript

// src/app/config.ts  
import { createConfig, http, cookieStorage, createStorage } from 'wagmi';  
import { mainnet, base, optimism } from 'wagmi/chains';  
  
export function getConfig() {  
 return createConfig({  
 chains: [mainnet, base, optimism],  
 ssr: true, // Critical: Enables server-side hydration support  
 storage: createStorage({  
 storage: cookieStorage, // Critical: Persists state in cookies accessible to the server  
 }),  
 transports: {  
 [mainnet.id]: http(),  
 [base.id]: http(),  
 [optimism.id]: http(),  
 },  
 });  
}

The ssr: true flag in Wagmi v2 signals the library to defer the initial hydration check, while cookieStorage provides the transport mechanism for the state. This combination eliminates the notorious UI flickering found in older Web3 apps.2

Root Layout Integration:

The root layout serves as the hydration entry point. It extracts the cookie from the incoming request headers and passes it to the cookieToInitialState helper.

TypeScript

// src/app/layout.tsx  
import { type ReactNode } from 'react';  
import { headers } from 'next/headers';  
import { cookieToInitialState } from 'wagmi';  
import { getConfig } from './config';  
import { Providers } from './providers'; // Client-side wrapper  
import './globals.css';  
  
export default async function RootLayout({ children }: { children: ReactNode }) {  
 // 1. Await headers (Next.js 15+ requirement, good practice in 14)  
 const headerList = await headers();  
 const cookie = headerList.get('cookie');  
   
 // 2. Hydrate state  
 const initialState = cookieToInitialState(getConfig(), cookie);  
  
 return (  
 <html lang="en">  
 <body>  
 <Providers initialState={initialState}>  
 {children}  
 </Providers>  
 </body>  
 </html>  
 );  
}

Provider Encapsulation:

Finally, we wrap the application in the WagmiProvider and QueryClientProvider. These must be in a client component (use client) because they use React Context.

TypeScript

// src/app/providers.tsx  
'use client';  
  
import { QueryClient, QueryClientProvider } from '@tanstack/react-query';  
import { useState, type ReactNode } from 'react';  
import { type State, WagmiProvider } from 'wagmi';  
import { getConfig } from './config';  
  
export function Providers({ children, initialState }: { children: ReactNode, initialState?: State }) {  
 const [config] = useState(() => getConfig());  
 const [queryClient] = useState(() => new QueryClient());  
  
 return (  
 <WagmiProvider config={config} initialState={initialState}>  
 <QueryClientProvider client={queryClient}>  
 {children}  
 </QueryClientProvider>  
 </WagmiProvider>  
 );  
}

This trio of files—config.ts, layout.tsx, and providers.tsx—creates a stable, hydration-safe foundation for the application.2

## 4. The Intelligence Layer: Modeling Intent

With the infrastructure in place, we move to the core of Project 1: the AI Intent Engine. This layer is responsible for parsing user inputs, identifying financial intents, and structuring them into executable data.

### 4.1 The Route Handler Architecture

We utilize a Next.js API Route (src/app/api/chat/route.ts) to host the Vercel AI SDK logic. This handler uses the streamText function, which allows us to stream the AI's thought process and tool calls back to the client in real-time.

Tool Definition Strategy:

In a Vibe Coding workflow, the definition of tools is the primary interface between the engineer and the AI. We use Zod schemas to strictly define the parameters of a DeFi transaction. The description field in Zod is not just documentation; it is part of the prompt that guides the LLM.

**Vibe Prompt 3: Route Handler Implementation**

"Create an API route at app/api/chat/route.ts. Import openai from @ai-sdk/openai and streamText from ai.

Define the system prompt: 'You are a DeFi Intent Agent. You analyze user requests to perform blockchain actions. If a user wants to swap, check balances, or bridge, call the appropriate tool. Be concise.'

Define a tool swapTokens with the following schema:

* tokenIn: string (symbol, e.g., 'ETH')
* tokenOut: string (symbol, e.g., 'USDC')
* amount: number (the quantity to swap)
* chain: string (e.g., 'base', 'mainnet')

Define a tool checkBalance with:

* token: string
* chain: string"

### 4.2 Implementation Logic

The code below demonstrates how to configure the streamText function to handle these tools. Note the use of convertToCoreMessages to sanitize the incoming message history.4

TypeScript

// src/app/api/chat/route.ts  
import { openai } from '@ai-sdk/openai';  
import { convertToCoreMessages, streamText } from 'ai';  
import { z } from 'zod';  
  
export const maxDuration = 30; // Timeout handling for long chains  
  
export async function POST(req: Request) {  
 const { messages } = await req.json();  
  
 const result = streamText({  
 model: openai('gpt-4o'),  
 system: `You are an advanced DeFi assistant.   
 - Help users execute transactions by generating UI components.   
 - When a user wants to swap, call 'swapTokens'.   
 - When a user asks for a balance, call 'checkBalance'.  
 - If the user's intent is ambiguous, ask for clarification.  
 - Always infer the chain if not specified, defaulting to Base.`,  
 messages: convertToCoreMessages(messages),  
 tools: {  
 swapTokens: {  
 description: 'Generate a UI for swapping tokens on a DEX.',  
 parameters: z.object({  
 tokenIn: z.string().describe('The token symbol to sell (e.g., ETH)'),  
 tokenOut: z.string().describe('The token symbol to buy (e.g., USDC)'),  
 amount: z.number().describe('The amount of tokenIn to swap'),  
 chain: z.enum(['mainnet', 'base', 'optimism']).default('base'),  
 }),  
 },  
 checkBalance: {  
 description: 'Check the wallet balance for a specific asset.',  
 parameters: z.object({  
 token: z.string().describe('The asset symbol'),  
 chain: z.enum(['mainnet', 'base', 'optimism']),  
 }),  
 },  
 },  
 });  
  
 return result.toDataStreamResponse();  
}

### 4.3 Multi-Step Reasoning and "Agentic Loops"

A key limitation of basic chatbots is their inability to handle dependencies. For example, "Check my ETH balance and then swap half of it for USDC." This requires two steps: first, a read operation; second, a calculation and a write intent.

The Vercel AI SDK supports this via the maxSteps configuration on the client. When maxSteps is set > 1, the client acts as a relay.

1. **Step 1:** AI calls checkBalance.
2. **Client:** Executes the read (via Wagmi readContract or an API) and sends the result back to the server.
3. **Step 2:** AI receives the balance (e.g., "10 ETH") and calculates "half" (5 ETH).
4. **Step 3:** AI calls swapTokens with amount: 5.

This loop transforms the interface from a command-line tool into a semi-autonomous agent.9

## 5. The Interface Layer: Generative UI

The Generative UI pattern decouples the *intent* (generated by the server) from the *execution* (rendered by the client). This ensures that sensitive operations, like signing transactions, happen securely in the user's browser context.

### 5.1 The useChat Hook and Tool Invocations

The useChat hook manages the conversation state. Crucially, it provides access to toolInvocations, an array of tool calls that the AI has generated. We iterate over this array to render the appropriate UI components.

**Vibe Prompt 4: Chat Interface**

"Create a ChatInterface component using useChat. Map over the messages.

* If the message is text, display it.
* If the message contains toolInvocations, switch on the toolName.
* For swapTokens, render a SwapCard component (I will create this later).
* For checkBalance, render a BalanceCard component.
* Handle the loading state where the tool is called but the result is not yet available."

### 5.2 Client-Side Rendering Logic

The following code demonstrates the rendering logic. Note how we handle the state of the tool invocation (result vs call).

TypeScript

// src/components/chat-interface.tsx  
'use client';  
  
import { useChat } from '@ai-sdk/react';  
import { SwapCard } from '@/components/tools/swap-card';  
import { BalanceCard } from '@/components/tools/balance-card';  
import { ScrollArea } from '@/components/ui/scroll-area';  
  
export default function ChatInterface() {  
 const { messages, input, handleInputChange, handleSubmit } = useChat({  
 maxSteps: 5, // Enable multi-step agentic behavior  
 });  
  
 return (  
 <div className="flex flex-col h-screen max-w-2xl mx-auto border-x">  
 <ScrollArea className="flex-1 p-4">  
 {messages.map((m) => (  
 <div key={m.id} className="mb-6">  
 {/\* 1. Render User/AI Text \*/}  
 <div className={`font-semibold mb-1 ${m.role === 'user'? 'text-blue-600' : 'text-gray-800'}`}>  
 {m.role === 'user'? 'You' : 'Agent'}  
 </div>  
 <div className="whitespace-pre-wrap text-gray-700">{m.content}</div>  
  
 {/\* 2. Render Generative UI Tools \*/}  
 <div className="mt-4 space-y-4">  
 {m.toolInvocations?.map((invocation) => {  
 const { toolName, toolCallId, args } = invocation;  
  
 if (toolName === 'swapTokens') {  
 return (  
 <SwapCard  
 key={toolCallId}  
 tokenIn={args.tokenIn}  
 tokenOut={args.tokenOut}  
 amount={args.amount}  
 chain={args.chain}  
 />  
 );  
 }  
   
 if (toolName === 'checkBalance') {  
 return (  
 <BalanceCard   
 key={toolCallId}   
 token={args.token}   
 chain={args.chain}   
 />  
 );  
 }  
 return null;  
 })}  
 </div>  
 </div>  
 ))}  
 </ScrollArea>  
  
 {/\* Input Area \*/}  
 <form onSubmit={handleSubmit} className="p-4 border-t bg-white">  
 <input  
 value={input}  
 onChange={handleInputChange}  
 placeholder="Type an intent (e.g., 'Swap 0.1 ETH for USDC')..."  
 className="w-full p-3 border rounded-lg shadow-sm focus:ring-2 focus:ring-blue-500"  
 />  
 </form>  
 </div>  
 );  
}

This implementation adheres to the research suggesting that mapping tool invocations on the client is superior to streaming RSCs for complex state management (like transaction flows).1

## 6. Component Implementation: The Swap Card

The SwapCard is the tangible manifestation of the user's intent. It is an interactive mini-application embedded in the chat stream. It must handle:

1. **Wallet Connection Check:** Is the user connected?
2. **Network Switching:** Is the user on the correct chain?
3. **Transaction Simulation:** Can this trade actually happen?
4. **Execution:** Signing and broadcasting the transaction.

### 6.1 The Wagmi Hooks

We utilize several Wagmi hooks to power this component:

* useAccount: Checks connection status.
* useSwitchChain: Enforces the correct network.
* useSendTransaction: The core execution hook.
* useWaitForTransactionReceipt: Monitors the blockchain for confirmation.

### 6.2 Detailed Component Architecture

The component is designed to be "optimistic" but editable. The AI suggests an amount, but the user can override it before signing. This "human-in-the-loop" design is critical for security.

**Vibe Prompt 5: Swap Card Logic**

"Create a SwapCard component.

* Props: tokenIn, tokenOut, amount, chain.
* UI: Use a Card with a header 'Swap Intent'. Display an input for the amount (pre-filled with the prop).
* Logic: Use useAccount. If disconnected, show a 'Connect Wallet' button.
* Transaction: Use useSendTransaction. For now, mock the transaction data.
* Feedback: Show a loading spinner during the transaction and a success message with the hash upon completion."

### 6.3 Production-Grade Code Example

Below is a refined implementation that integrates Shadcn components with Wagmi logic.

TypeScript

// src/components/tools/swap-card.tsx  
'use client';  
  
import { useState } from 'react';  
import { useAccount, useSendTransaction, useWaitForTransactionReceipt, useSwitchChain } from 'wagmi';  
import { parseEther } from 'viem';  
import { Card, CardHeader, CardTitle, CardContent, CardFooter } from '@/components/ui/card';  
import { Button } from '@/components/ui/button';  
import { Input } from '@/components/ui/input';  
import { Loader2, CheckCircle } from 'lucide-react';  
  
interface SwapCardProps {  
 tokenIn: string;  
 tokenOut: string;  
 amount: number;  
 chain: string; // e.g., 'base'  
}  
  
export function SwapCard({ tokenIn, tokenOut, amount: initialAmount, chain }: SwapCardProps) {  
 const [amount, setAmount] = useState(initialAmount.toString());  
 const { isConnected, chainId } = useAccount();  
 const { switchChain } = useSwitchChain();  
   
 // Transaction Hooks  
 const { data: hash, sendTransaction, isPending, error } = useSendTransaction();  
 const { isLoading: isConfirming, isSuccess } = useWaitForTransactionReceipt({ hash });  
  
 // Mock Router Address (Uniswap Universal Router on Base)  
 const ROUTER\_ADDRESS = '0x3fC91A3afd70395Cd496C647d5a6CC9D4B2b7FAD';   
  
 const handleSwap = () => {  
 if (!isConnected) return; // Trigger wallet modal here in real app  
   
 // In a real scenario, you would fetch encoded calldata from an API (e.g., 1inch/Uniswap)  
 // Here we perform a basic transfer to simulate interaction  
 sendTransaction({  
 to: ROUTER\_ADDRESS,  
 value: parseEther(amount),  
 // data: '0x...' // Encoded swap data would go here  
 });  
 };  
  
 return (  
 <Card className="w-full border-2 border-slate-100 shadow-md">  
 <CardHeader className="pb-2">  
 <CardTitle className="text-sm font-medium text-slate-500 uppercase tracking-wider">  
 AI Suggested Action  
 </CardTitle>  
 </CardHeader>  
   
 <CardContent className="space-y-4">  
 {/\* Token Input Section \*/}  
 <div className="flex items-center space-x-2 p-3 bg-slate-50 rounded-lg">  
 <Input   
 type="number"   
 value={amount}   
 onChange={(e) => setAmount(e.target.value)}  
 className="border-none bg-transparent text-2xl font-bold focus-visible:ring-0"  
 />  
 <div className="flex flex-col items-end">  
 <span className="font-bold">{tokenIn}</span>  
 <span className="text-xs text-slate-400">Balance: Loading...</span>  
 </div>  
 </div>  
  
 <div className="flex justify-center -my-2 relative z-10">  
 <div className="bg-white p-1 rounded-full border">  
 <span className="text-slate-400">⬇️</span>  
 </div>  
 </div>  
  
 {/\* Token Output Section \*/}  
 <div className="flex items-center justify-between p-3 bg-slate-50 rounded-lg">  
 <span className="text-2xl font-bold text-slate-400">~</span>  
 <span className="font-bold">{tokenOut}</span>  
 </div>  
 </CardContent>  
  
 <CardFooter className="pt-2">  
 {!isConnected? (  
 <Button className="w-full" variant="outline">Connect Wallet</Button>  
 ) : isSuccess? (  
 <Button className="w-full bg-green-500 hover:bg-green-600 cursor-default">  
 <CheckCircle className="mr-2 h-4 w-4" /> Swap Complete  
 </Button>  
 ) : (  
 <Button   
 className="w-full"   
 onClick={handleSwap}   
 disabled={isPending |  
  
| isConfirming}  
 >  
 {isPending? <Loader2 className="animate-spin mr-2" /> : null}  
 {isPending? 'Check Wallet...' : isConfirming? 'Confirming...' : 'Confirm Swap'}  
 </Button>  
 )}  
 </CardFooter>  
   
 {/\* Error Handling Display \*/}  
 {error && (  
 <div className="px-6 pb-4 text-xs text-red-500">  
 Error: {error.message.slice(0, 50)}...  
 </div>  
 )}  
 </Card>  
 );  
}

This component illustrates the "Generative UI" philosophy: the AI sets the initial state (via props), but the React component manages the transaction lifecycle (via Wagmi). This ensures that the user remains in control of their assets.11

## 7. Advanced Integration: Agentic Loops & Safety

### 7.1 Multi-Step Interactions (The Agentic Loop)

To enable true intent satisfaction, we implement the "Agentic Loop." This is where the output of a client-side tool is fed back to the AI.

For example, if the user asks "Swap all my ETH," the AI doesn't know how much ETH the user has.

1. **AI:** Calls checkBalance({ token: 'ETH' }).
2. **Client:** The ChatInterface detects this tool. It executes useReadContract to fetch the balance.
3. **Loop:** The client *automatically* calls addToolResult (from the Vercel AI SDK) with the value "1.5 ETH".
4. **AI:** Receives "1.5 ETH". It then generates the swapTokens tool call with amount: 1.5.

This requires the useChat hook's maxSteps property to be set to a value greater than 1 (e.g., 5).

### 7.2 Safety & Sanitization

Vibe coding facilitates rapid development, but security protocols must be explicit.

* **Prompt Injection:** The system prompt must explicitly forbid the AI from executing tools that transfer assets to arbitrary addresses provided in the user prompt (e.g., "Send all money to 0x..."). The swapTokens tool should strictly interact with known Router contracts.
* **Zod Validation:** The tokenIn and tokenOut fields should be validated against a whitelist of supported tokens (e.g., a token map) inside the tool's execute function or on the client side to prevent interactions with malicious contracts.

## 8. Development Workflow & Testing

### 8.1 The "Mock" Connector Strategy

Testing Web3 apps is painful because it usually requires spending real gas or setting up local forks (Anvil/Hardhat). For Vibe Coding, we use Wagmi's **Mock Connector**. This allows us to simulate connection states, transaction delays, and errors without a real wallet.

**Vibe Prompt 6: Mock Setup**

"Modify config.ts. If process.env.NODE\_ENV === 'development', add a mock connector from wagmi/connectors. Configure it with a fake account address. This will allow me to vibe code the 'Success' states of the UI without approving transactions in Metamask.".13

### 8.2 Troubleshooting Common Issues

| **Issue** | **Cause** | **Solution** |
| --- | --- | --- |
| **Hydration Error** | Client wallet state differs from Server default. | Ensure cookieStorage is configured and cookieToInitialState is used in layout.tsx. |
| **Flickering UI** | React Server Components remounting on stream updates. | Use streamText (data) instead of streamUI (components) and map via useChat. |
| **"Invalid Hook Call"** | Wagmi hooks used inside async Server Components. | Ensure all components using useAccount or useSendTransaction are marked 'use client'. |
| **AI Hallucination** | System prompt too vague. | Use strict Zod schemas with describe() fields and enum constraints for chains/tokens. |

## Conclusion

"Project 1: The Natural Language DeFi Intent Interface" represents a convergence of modern web development patterns. By harmonizing the **Next.js App Router** for infrastructure, the **Vercel AI SDK** for intent modeling, and **Wagmi v2** for blockchain state, we create an experience that is both magical and secure.

The **Vibe Coding** methodology outlined here—reliant on iterative prompting, strict schema definition, and component-based generative UI—allows developers to transcend the complexities of Web3 boilerplate. The result is not just a chatbot, but a functional financial agent capable of understanding user intent and scaffolding the necessary cryptographic transactions to fulfill it. This architecture provides a robust, production-ready foundation for the next generation of DeFi applications.

#### Works cited

1. Migrating from RSC to UI - AI SDK, accessed on December 31, 2025, <https://ai-sdk.dev/docs/ai-sdk-rsc/migrating-to-ui>
2. SSR | Wagmi, accessed on December 31, 2025, <https://wagmi.sh/react/guides/ssr>
3. Getting Started: Next.js App Router - AI SDK, accessed on December 31, 2025, <https://ai-sdk.dev/docs/getting-started/nextjs-app-router>
4. AI SDK 5 - Vercel, accessed on December 31, 2025, <https://vercel.com/blog/ai-sdk-5>
5. Real-time AI in Next.js: How to stream responses with the Vercel AI SDK - LogRocket Blog, accessed on December 31, 2025, <https://blog.logrocket.com/nextjs-vercel-ai-sdk-streaming/>
6. Chatbot Tool Usage - AI SDK UI, accessed on December 31, 2025, <https://ai-sdk.dev/docs/ai-sdk-ui/chatbot-tool-usage>
7. Next.js - Shadcn UI, accessed on December 31, 2025, <https://ui.shadcn.com/docs/installation/next>
8. Adding Your First Component | Vercel Academy, accessed on December 31, 2025, <https://vercel.com/academy/shadcn-ui/adding-your-first-component>
9. Introducing Vercel AI SDK 3.2, accessed on December 31, 2025, <https://vercel.com/blog/introducing-vercel-ai-sdk-3-2>
10. Multi-Step & Generative UI | Vercel Academy, accessed on December 31, 2025, <https://vercel.com/academy/ai-sdk/multi-step-and-generative-ui>
11. Send Transaction | Wagmi, accessed on December 31, 2025, <https://wagmi.sh/react/guides/send-transaction>
12. Send Transaction - React Hooks for Ethereum - Wagmi, accessed on December 31, 2025, <https://1.x.wagmi.sh/examples/send-transaction>
13. Mock – wagmi, accessed on December 31, 2025, <https://1.x.wagmi.sh/react/connectors/mock>
14. How to mock connector with @wagmi/core v2 #3420 - GitHub, accessed on December 31, 2025, <https://github.com/wevm/wagmi/discussions/3420>