The Architect's Handbook to ChatGPT Application Development: SDKs, Rendering Protocols, and Generative UI Systems

Executive Summary

The emergence of Large Language Models (LLMs) has necessitated a fundamental reimagining of software architecture. We are transitioning from a deterministic paradigm—where user interfaces (UIs) are static and interactions are explicitly programmed—to a probabilistic paradigm where interfaces are generated, routed, and rendered dynamically based on natural language inference. This shift has bifurcated the development landscape into two distinct but overlapping ecosystems: the **internal** ecosystem of applications that reside within the ChatGPT interface (governed by the OpenAI Apps SDK and Model Context Protocol) and the **external** ecosystem of standalone AI applications (governed by libraries such as the Vercel AI SDK).

This report serves as a definitive technical reference for software architects and frontend engineers tasked with navigating this complex terrain. It provides an exhaustive analysis of the mechanisms underpinning UI implementation, rendering pipelines, state management, and stream processing. Unlike superficial tutorials, this document deconstructs the low-level protocols—from the byte-level parsing of Server-Sent Events (SSE) to the iframe communication bridges of sandboxed widgets—enabling developers to build robust, production-grade Generative UI systems. We will explore the Model Context Protocol (MCP) as the standardized connective tissue for AI tools, the rigorous security constraints of the Apps SDK, and the sophisticated client-side orchestration capabilities of the Vercel AI SDK. By synthesizing data from official documentation, community patterns, and low-level debugging logs, this report offers a nuanced, cohesive narrative on the state of the art in AI application development.

Part I: The Theoretical Framework of Generative Applications

To understand the specific implementations of the OpenAI Apps SDK or Vercel AI SDK, one must first grasp the theoretical shift they represent. Traditional software relies on "Imperative UI," where the developer defines exactly what screen appears when a button is clicked. The new paradigm is "Generative UI" or "Intent-Based UI," where the system interprets a user's intent and constructs a bespoke interface to satisfy it.

1.1 From Static Plugins to Interactive Pixels

The lineage of the current ecosystem begins with the now-deprecated OpenAI Plugins. Plugins were purely API bridges; the model could query a weather API and output text, but it could not render a weather map. The user experience was fragmented—rich reasoning coupled with poor presentation.

The introduction of the **ChatGPT Apps SDK** marks the maturity of this concept. It introduces the ability to render "interactive pixels"—custom web components—directly within the chat stream.[1] This is not merely an aesthetic upgrade; it is a functional revolution. It allows for "human-in-the-loop" workflows where the model proposes a complex action (e.g., a SaaS architecture diagram or a multi-leg flight itinerary), and the user manipulates a graphical widget to refine that proposal before finalizing it.[1] The "interactive pixels" capability transforms ChatGPT from a text generator into a multi-modal application runtime, where the interface itself is ephemeral and context-dependent.

1.2 The Model Context Protocol (MCP) Standard

Underpinning this new runtime is the **Model Context Protocol (MCP)**. MCP is an open standard designed to standardize how AI models interact with external data and tools. In the previous "Plugins" era, every integration required a custom OpenAPI specification. MCP normalizes this into a server-client architecture where the "Server" (the developer's backend) exposes capabilities, and the "Client" (the LLM host, e.g., ChatGPT) consumes them.[3]

The MCP architecture is critical because it decouples the *definition* of a tool from its

presentation. An MCP server defines a tool like get_stock_price. In a text-only client (like a terminal), this returns JSON. In a rich client (like ChatGPT with the Apps SDK), the same tool definition carries metadata that triggers the rendering of a financial chart widget. This separation of concerns allows developers to build "Headless AI" logic that can be skinned differently depending on the consuming platform.[5]

1.3 The Generative UI Spectrum

We can categorize current development patterns into a spectrum of Generative UI:

- **Level 1: Text-Based Tool Use:** The model calls a tool, gets data, and summarizes it in text (Standard ChatGPT).
- **Level 2: Client-Side Component Mapping:** The model returns a structured "Tool Call," and the client-side code maps this intent to a pre-built React component (Vercel AI SDK useChat pattern).
- **Level 3: Server-Side Component Streaming:** The server generates the actual UI component code (or a serialized tree) and streams it to the client, allowing for interfaces the client code never anticipated (Vercel AI SDK RSC pattern).
- **Level 4: Sandboxed Widget Injection:** The host injects a secure, isolated iframe containing a mini-application provided by the server (OpenAI Apps SDK pattern).

This report will analyze Levels 2, 3, and 4 in exhaustive detail, as they represent the frontier of current development.

Part II: Deep Dive into The OpenAI Apps SDK

The OpenAI Apps SDK is the framework for building applications that live *inside* ChatGPT. It differs fundamentally from building a custom chatbot; here, you are a guest in OpenAI's house. You must adhere to strict protocols regarding security, rendering, and state management. The architecture relies on three pillars: The MCP Server, The Widget Bundle, and the Window Bridge.

2.1 The MCP Server Architecture

The backend of a ChatGPT App is an MCP Server. This server is responsible for defining the "surface area" of your application—the tools and resources available to the model.

2.1.1 Tool Definition and The _meta Protocol

In standard MCP, a tool is defined by its name, description, and input schema (usually Zod-based). In the Apps SDK, the tool definition is augmented with a critical _meta field. This field is the "instruction manual" for the ChatGPT frontend, telling it how to visualize the tool's result.

The _meta object is invisible to the LLM's context window—an important optimization to save tokens—but it is parsed by the host to determine rendering behavior.[6]

Key _meta Fields:

Field Key	Type	Purpose	Implications
openai/outputTemplate	string (URI)	Specifies the resource URI of the HTML widget (e.g., ui://widget/dashboard.html).	This is mandatory for triggering a UI render. Without it, the model only sees JSON.[8]
openai/widgetAccessible	boolean	Permissions flag allowing the widget to call tools via the bridge.	Defaults to false. Must be true for interactive widgets (e.g., a "Save" button in the UI).[8]
openai/toolInvocation/invoking	string	Status text displayed while the tool runs (e.g., "Searching database...").	Improves perceived latency. Max 64 chars.[8]
openai/toolInvocation/invoked	string	Status text displayed after completion (e.g.,	Provides closure to the user before the widget appears.[8]

		"Dashboard ready").	
securitySchemes	array	Defines authentication requirements (e.g., OAuth scopes).	Mirrors the standard MCP security but specifically for the host's auth flow.[8]

Architectural Insight: The separation of structuredContent (the actual data returned by the tool) and _meta (the rendering instructions) is a deliberate architectural choice. It allows the model to reason about the *data* (e.g., "The stock price is $150, so I should recommend selling") without being confused by the *presentation* details (HTML templates or CSP rules). This ensures the reasoning capability remains robust even as the UI evolves.[9]

2.1.2 Resource Registration and the ui:// Scheme

Resources in MCP are typically static files (logs, docs). In the Apps SDK, resources are the delivery mechanism for the frontend code. Developers register resources using a custom ui:// URI scheme.

When the MCP server registers a tool with "openai/outputTemplate": "ui://widget/todo.html", it must also register a corresponding resource for that URI.

TypeScript

```
// Conceptual Node.js MCP Server Setup
server.registerResource(
  "todo-widget",
  "ui://widget/todo.html", // The logical URI
  { mimeType: "text/html" }, // MIME type is critical
  async () => ({
    contents:
  })
);
```

This mapping enables the ChatGPT client to fetch the UI bundle securely from the MCP

connection without exposing a public internet URL, maintaining the security boundary of the session.[2]

2.2 The Widget Rendering Pipeline

The rendering of a widget is a multi-step orchestration between the Model, the MCP Server, and the ChatGPT Client (Host). Understanding this flow is essential for debugging "blank screen" or "loading loop" issues.[12]

1. **Intent Recognition:** The user prompts, "Show my tasks." The model selects the get_tasks tool.
2. **Tool Execution:** The MCP server executes get_tasks. It returns a payload containing structuredContent (the list of tasks) and _meta (pointing to ui://widget/list.html).
3. **Template Resolution:** The Host sees the outputTemplate and requests the resource ui://widget/list.html from the MCP server.
4. **Sandbox Initialization:** The Host spins up a sandboxed <iframe> on a randomized subdomain (e.g., usercontent.chatgpt.com).
5. **Injection & Hydration:** The Host injects the HTML/JS bundle into the iframe. Crucially, it also injects the structuredContent directly into the window.openai global object.
6. **Rendering:** The React (or vanilla JS) code in the widget reads window.openai.toolOutput and renders the initial state.[10]

Critical Troubleshooting Note: A common failure mode occurs when build tools (like Vite or Webpack) append hashes to filenames (e.g., index-2d2b.js) for cache busting. If the MCP server is configured to serve index.js but the HTML file references index-2d2b.js, the widget will fail to load in the sandbox. Developers must ensure canonical filenames are used in the MCP resource registration or that the HTML template is dynamically updated to match the build artifacts.[14]

2.3 The window.openai Bridge API

Inside the sandboxed iframe, the widget is cut off from the outside world. It cannot access the parent window's DOM or arbitrary cookies. Its only lifeline is the window.openai object. This API serves as the bridge for data ingestion, state management, and communicating back to the host.[16]

2.3.1 Data Ingestion: toolOutput

The primary data source for a widget is window.openai.toolOutput. This object contains the structuredContent returned by the tool.

- **Pattern:** In React, one should use a hook to subscribe to this global. It is not static; it can change if the model calls the tool again or if the user triggers a refresh.
- **Type Safety:** Developers should define a TypeScript interface for their ToolOutput to ensure the frontend code aligns with the Zod schema defined on the backend.[5]

TypeScript

```
// Example: Safely consuming tool output
const toolOutput = useOpenAiGlobal('toolOutput');
const tasks = toolOutput?.result?.structuredContent?.tasks ||;
```

2.3.2 State Management: widgetState and setWidgetState

Widgets are ephemeral. If a user scrolls up, the iframe might be unloaded to save memory. If they reload the page, the iframe is destroyed. Therefore, standard React useState is insufficient for persisting user interactions (e.g., a "tab selection" or "form draft").

The SDK provides a mechanism to persist state *in the conversation history* via setWidgetState.

- **Mechanism:** When window.openai.setWidgetState({ currentTab: 'details' }) is called, the host saves this JSON payload associated with the specific message ID.
- **Rehydration:** When the widget reloads, window.openai.widgetState is pre-populated with { currentTab: 'details' }.
- **Latency:** This operation is asynchronous and involves communication with the OpenAI backend. It should be treated like a database write, not a local variable update.[19]

Best Practice: Use optimistic UI updates. Update the local React state immediately for responsiveness, then call setWidgetState to persist it.

2.3.3 The Follow-Up Loop: sendFollowUpMessage

Widgets are not just displays; they are control surfaces. A "Delete" button in a widget cannot directly delete a record in the database because the widget is sandboxed. Instead, it must instruct the *model* to perform the action.

This is done via window.openai.sendFollowUpMessage({ prompt: "Delete task ID 123" }).

1. User clicks "Delete".
2. Widget calls sendFollowUpMessage.
3. This appears as a user message in the chat (invisible or visible depending on configuration).
4. The model interprets this prompt.
5. The model calls the delete_task tool on the MCP server.
6. The tool executes and returns the new list of tasks.
7. The widget receives the new data via toolOutput and re-renders.

This "Prompt-Driven Interaction" loop ensures that the model remains the central orchestrator of all side effects, maintaining safety and alignment.[18]

2.4 Security: Content Security Policy (CSP)

The Apps SDK enforces a rigorous Content Security Policy to prevent data exfiltration. By default, widgets cannot make network requests to arbitrary domains.

Developers must declare a widgetCSP in the _meta field of their resource.

- connect_domains: A whitelist of domains the widget can fetch() from (e.g., your API backend).
- resource_domains: A whitelist of domains for loading images, fonts, or scripts (e.g., fonts.googleapis.com).

Troubleshooting: If a widget displays a "Network Error" or fails to load an image, the first step is to check the browser console for CSP violations. The sandbox will explicitly block any domain not listed in this metadata.[21]

Part III: Deep Dive into The Vercel AI SDK

While the OpenAI Apps SDK allows you to build *in* ChatGPT, the Vercel AI SDK is the gold standard for building *your own* AI platforms using OpenAI's models (and others). It provides the plumbing for streaming, tool calling, and rendering in external environments like Next.js or Nuxt.[23]

3.1 The Architecture of useChat

The core abstraction of the Vercel AI SDK is the useChat hook. It manages the entire lifecycle of a chat session:

1. **Optimistic UI:** Immediately adds the user's message to the state.
2. **Streaming Transport:** Initiates a POST request to the API route.
3. **Stream Parsing:** Decodes the incoming stream (which may contain text, tool calls, or data).
4. **State Reconciliation:** Updates the messages array incrementally, triggering re-renders at 60fps to create the "typing" effect.[23]

3.2 Client-Side Tool Calling: The onToolCall Pattern

A major evolution in Vercel AI SDK v5 is the support for client-side tools. Previously, tools were strictly server-side (e.g., querying a database). Now, the LLM can "control" the user's browser.[26]

Scenario: The user asks, "Where am I?" The model wants to use the browser's Geolocation API. This cannot happen on the server.

Implementation Pattern:
In useChat, the developer defines an onToolCall handler.

TypeScript

```
// app/page.tsx (Client Component)
const { messages } = useChat({
  async onToolCall({ toolCall }) {
    if (toolCall.toolName === 'getLocation') {
      // Execute browser-only logic
      const position = await getBrowserLocation();

      // Return result to the chat history
      // WARNING: Do not await addToolResult if it causes a re-render loop
      return position;
    }
  }
});
```

When the model invokes getLocation, the SDK halts the stream, triggers this callback, waits for the result, sends the result back to the server (via addToolResult), and then resumes the stream to generate the final response.[26]

Deadlock Warning: A known issue exists where awaiting addToolResult inside certain event loops can cause a deadlock. It is often safer to fire the result submission as a side effect or ensure the tool logic is strictly isolated from the rendering cycle.[29]

3.3 The "Tool-to-UI" Mapping Pattern

Unlike the Apps SDK, which uses iframes, the Vercel AI SDK renders tools directly in the main DOM. This requires a mapping strategy in the rendering loop.

The message object contains a parts array (in v5) or toolInvocations (in v4). The rendering loop iterates over these parts to decide what to show.

TypeScript

```
// Rendering Loop in JSX
{message.parts.map((part) => {
  if (part.type === 'text') return <Markdown>{part.text}</Markdown>;

  if (part.type === 'tool-invocation') {
```

```
    const { toolName, state, result } = part.toolInvocation;

    // Probabilistic Routing
    if (toolName === 'showTicker') {
      if (state === 'result') return <StockTicker data={result} />;
      return <SkeletonLoader />;
    }
  }
})}
```

This pattern allows for seamless transitions between text bubbles and interactive components. The "Skeleton" state is critical for perceived performance, giving the user feedback while the tool executes.[25]

3.4 Generative UI via React Server Components (RSC)

The Vercel AI SDK also supports a more advanced paradigm called "Generative UI" using React Server Components (RSC). In the client-side pattern (3.3), the client must have the StockTicker component imported and bundled. In the RSC pattern, the server *streams the component code itself*.[32]

Using the streamUI function on the server:

1. The model calls a tool.
2. The server executes the tool.
3. The server returns a ReactNode (e.g., <StockChart symbol="AAPL" />).
4. This node is serialized and streamed to the client.

Comparison:

- **Client-Side Mapping:** Better for apps with a fixed set of known widgets (e.g., a dashboard). Faster interactivity since code is already loaded.
- **RSC Generative UI:** Better for highly dynamic apps where the variety of possible UIs is vast. The initial load might be slower (streaming markup), but it allows for infinite flexibility without bloating the initial client bundle.[33]

Part IV: The Physics of Streaming

For developers building custom implementations or debugging SDK behavior, understanding the low-level physics of the data stream is mandatory. The "magic" of ChatGPT is just a rigorous application of the Server-Sent Events (SSE) protocol.

4.1 Anatomy of the Byte Stream

When stream=true is set in the OpenAI API, the response is not JSON. It is a continuous flow of bytes. A parser must read this stream using a ReadableStreamDefaultReader.[35]

The Parsing Algorithm:

1. **Read:** reader.read() returns a Uint8Array chunk.
2. **Decode:** new TextDecoder().decode(chunk) converts bytes to a string.
3. **Buffer:** The chunk might end in the middle of a JSON object. You must buffer the string until a double newline (\n\n) is detected, which delimits messages in SSE.
4. **Filter:** Each line starts with data: . This prefix must be stripped. The string `` indicates the end of the stream.
5. **Parse:** The remaining string is parsed as JSON.

4.2 Handling Azure OpenAI Quirks

Developers utilizing Azure OpenAI often encounter crashes because Azure injects proprietary "Content Filter" chunks into the stream. These chunks (containing prompt_filter_results) often lack the choices array found in standard OpenAI chunks.

Debugging Insight: A robust parser must conservatively check for the existence of chunk.choices before accessing chunk.choices.delta. Failing to do so results in "Cannot read property 'delta' of undefined" errors, a common cause of white-screen crashes in production apps using Azure.[37]

4.3 Reconstructing Tool Call Deltas

Streaming tool calls is the most complex part of the protocol. The arguments for a function

(e.g., {"location": "Paris"}) are streamed character by character.

The parser must implement a state machine:

1. **Index Tracking:** Tool calls have an index. The parser must track which tool call index it is currently accumulating.
2. **Accumulation:** It must append the function.arguments delta to a buffer string for that specific index.
3. **Validation:** It cannot JSON.parse() the buffer until the finish_reason is received, as {"loca is not valid JSON.

The "Flashing" Artifact: If you try to parse and render the tool call args in real-time (to show a progress bar, for instance), you must use a "partial JSON parser" that can handle incomplete strings, otherwise, the console will be flooded with syntax errors.[38]

Part V: Advanced Implementation: The "Pizza App" Pattern

To synthesize these concepts, we detail the implementation of a production-grade "Pizza Ordering" app using the OpenAI Apps SDK. This follows the pattern referenced in research but expands with full technical context.[12]

5.1 The MCP Server Definition

We define an MCP server that exposes an order_pizza tool.

TypeScript

```
// server.ts
const server = new McpServer({ name: "PizzaBot", version: "1.0.0" });

server.registerTool("order_pizza",
 {
```

```
  inputSchema: z.object({
    type: z.string().describe("Type of pizza (e.g. Pepperoni)"),
    size: z.string().describe("Size: S, M, L")
  }),
  _meta: {
    // Point to the UI resource
    "openai/outputTemplate": "ui://widget/order-card.html",
    "openai/toolInvocation/invoking": "Firing up the oven...",
    "openai/toolInvocation/invoked": "Order Ticket Created."
  }
 },
 async ({ type, size }) => {
  const orderId = `ORD-${Math.floor(Math.random() * 10000)}`;
  // The model sees this text
  const content = `Created order ${orderId} for a ${size} ${type} pizza.`;

  // The widget sees this structured object
  const structuredContent = {
    orderId,
    status: 'confirmed',
    details: { type, size },
    trackingUrl: `/api/track/${orderId}`
  };

  return { content, structuredContent };
 }
);
```

5.2 The Widget Resource

We must register the HTML file that corresponds to ui://widget/order-card.html.

TypeScript

```
// server.ts (continued)
server.registerResource(
```

```
  "order-card-resource",
  "ui://widget/order-card.html",
  { mimeType: "text/html" },
  async () => ({
    contents:
  })
);
```

5.3 The Interactive Widget (Client Code)

Inside public/widget.html, we build the UI using React (via CDN for simplicity in this example, though bundling is recommended for production).

HTML

```
<script type="module">
  import { useState, useEffect } from 'https://esm.sh/react';
  import { createRoot } from 'https://esm.sh/react-dom/client';

  function PizzaWidget() {
    // 1. Ingest Data
    const initialData = window.openai.toolOutput;
    const = useState(initialData.status);

    // 2. Handle Follow-Up Action
    const handleTrack = () => {
      // Instruct the model to perform the next step
      window.openai.sendFollowUpMessage({
        prompt: `Check status of order ${initialData.orderId}`
      });
    };

    return (
      <div className="card">
        <h2>Order {initialData.orderId}</h2>
        <div className="badge">{status}</div>
        <p>{initialData.details.size} {initialData.details.type}</p>
        <button onClick={handleTrack}>Track Delivery</button>
      </div>
```

```
  );
 }

 const root = createRoot(document.getElementById('root'));
 root.render(<PizzaWidget />);
</script>
```

This interaction loop—User prompts -> Model calls Tool -> Server returns Data + Template -> Widget renders -> User clicks -> Widget sends Follow-Up—is the canonical cycle of a ChatGPT App.

Part VI: Troubleshooting & Maintenance

Deploying these systems involves navigating a web of potential failures. This section catalogs common errors and their resolutions.

6.1 Common Error Signatures

Error Message	Context	Root Cause	Resolution
"Error in body stream"	Vercel AI SDK	Network interruption or timeout during streaming.	Implement retry logic with exponential backoff. Check server timeout settings (e.g., Vercel's 10s serverless limit).[40]
"AuthenticationError" / 401	OpenAI API	Invalid or expired API key.	Rotate keys. Ensure environment variables (OPENAI_API_KEY) are correctly loaded in the

			runtime environment.[40]
"Cannot read properties of undefined (reading 'tool_calls')"	Stream Parsing	Azure OpenAI sending filter chunks without choices.	Update parser to check chunk.choices?.length > 0 before accessing properties.[37]
Widget fails to load (404)	Apps SDK	Hashed filenames in build (e.g., app-5f2a.js) not matching the registered resource URI.	Disable filename hashing in your bundler (Vite/Webpack) or dynamically update the resource registration to match the build manifest.[14]
"Network Error" inside Widget	Apps SDK	Content Security Policy (CSP) violation.	Add the target domain to the connect_domains or resource_domains list in the tool's _meta definition.[22]

6.2 Debugging Strategies

The MCP Inspector:
OpenAI and the MCP community provide an "MCP Inspector" tool. This allows developers to test their MCP server locally without connecting to ChatGPT. It provides a UI to invoke tools and see the raw JSON response, including the _meta fields. This is indispensable for verifying that outputTemplate URIs are correct before deployment.21
The window.openai Logger:
Since standard console.log inside an iframe can be hard to see, a common pattern is to create a debug area within the widget itself that prints the contents of window.openai.toolOutput and window.openai.widgetState. This confirms whether data injection is happening correctly.

Part VII: Strategic Implications and Future Outlook

The convergence of the OpenAI Apps SDK and Vercel AI SDK signals a maturing of the Generative UI landscape.

Strategic Selection:

- **Use OpenAI Apps SDK if:** Your goal is distribution. You want to reach the 200M+ users of ChatGPT directly. Your app is a utility that enhances the chat experience (e.g., a diagram generator, a specialized calculator).
- **Use Vercel AI SDK if:** Your goal is product ownership. You are building a standalone SaaS where AI is a feature. You need full control over the UI, authentication, and billing outside of OpenAI's ecosystem.[41]

The Hybrid Future:
We are observing the emergence of "Hybrid Architectures." Developers build a core MCP server that encapsulates their business logic. This server is then connected to both ChatGPT (via the Apps SDK) and a custom Next.js frontend (via an MCP client adapter). This maximizes leverage: write the tool once, render it everywhere.
As models become faster and cheaper, the latency of "Generative UI"—where the interface is drawn in real-time—will approach that of static UI. The rigid distinction between "content" and "application" is dissolving. In this new era, the prompt is the URL, the model is the router, and the generated widget is the page.

Works cited

1. Everything you need to know about building ChatGPT apps - Gadget.dev, accessed November 25, 2025, https://gadget.dev/blog/everything-you-need-to-know-about-building-chatgpt-apps
2. ChatGPT Apps SDK: Proven 2025 Guide For Building Apps - Binary Verse AI, accessed November 25, 2025, https://binaryverseai.com/chatgpt-apps-sdk-guide-build-apps-tutorial/
3. OpenAI's Apps SDK: A Developer's Guide to Getting Started - The New Stack, accessed November 25, 2025, https://thenewstack.io/openais-apps-sdk-a-developers-guide-to-getting-started/
4. Building MCP servers for ChatGPT and API integrations - OpenAI Platform, accessed November 25, 2025, https://platform.openai.com/docs/mcp
5. Building Custom Tools and Widgets for ChatGPT: The OpenAI Apps SDK Handbook | by Sumit Paul | Oct, 2025 | Medium, accessed November 25, 2025,

https://medium.com/@sumit-paul/building-custom-tools-and-widgets-for-chatgpt-the-openai-apps-sdk-handbook-fef47a7ba555
6. Define tools - OpenAI for developers, accessed November 25, 2025, https://developers.openai.com/apps-sdk/plan/tools/
7. Build your MCP server - OpenAI for developers, accessed November 25, 2025, https://developers.openai.com/apps-sdk/build/mcp-server/
8. Reference - OpenAI for developers, accessed November 25, 2025, https://developers.openai.com/apps-sdk/reference/
9. Show us what you're building with the ChatGPT Apps SDK - OpenAI Developer Community, accessed November 25, 2025, https://community.openai.com/t/show-us-what-you-re-building-with-the-chatgpt-apps-sdk/1365862
10. OpenAI Apps SDK Quickstart: Build Your First ChatGPT App in Python in 15 Minutes, accessed November 25, 2025, https://mcpcat.io/guides/openai-apps-sdk-quickstart-python/
11. Quickstart - OpenAI for developers, accessed November 25, 2025, https://developers.openai.com/apps-sdk/quickstart/
12. How to Use the ChatGPT Apps SDK: Build a Pizza App with Apps SDK - freeCodeCamp, accessed November 25, 2025, https://www.freecodecamp.org/news/how-to-use-the-chatgpt-apps-sdk/
13. Inside OpenAI's Apps SDK: how to build interactive ChatGPT apps with MCP - Alpic AI, accessed November 25, 2025, https://alpic.ai/blog/inside-openai-s-apps-sdk-how-to-build-interactive-chatgpt-apps-with-mcp?ref=makerswave.com
14. No widget UI showing · Issue #53 · openai/openai-apps-sdk-examples - GitHub, accessed November 25, 2025, https://github.com/openai/openai-apps-sdk-examples/issues/53
15. App UI / Widget via AppsSDK is not rendering since 16th October 2025, 20:00 IST #62, accessed November 25, 2025, https://github.com/openai/openai-apps-sdk-examples/issues/62
16. Build your ChatGPT UI - OpenAI for developers, accessed November 25, 2025, https://developers.openai.com/apps-sdk/build/chatgpt-ui/
17. Build your ChatGPT UI - OpenAI for developers, accessed November 25, 2025, https://developers.openai.com/apps-sdk/build/chatgpt-ui
18. From Announcement to Implementation: Building ChatGPT Apps | TELUS Digital, accessed November 25, 2025, https://www.telusdigital.com/insights/data-and-ai/article/building-chatgpt-apps
19. Managing State - OpenAI for developers, accessed November 25, 2025, https://developers.openai.com/apps-sdk/build/state-management/
20. Build a Custom ChatGPT App and Tap Into 800 Million Users - Hackernoon, accessed November 25, 2025, https://hackernoon.com/build-a-custom-chatgpt-app-and-tap-into-800-million-users
21. Guide to authentication and user consent in the OpenAI Apps SDK - Stytch, accessed November 25, 2025,

https://stytch.com/blog/guide-to-authentication-for-the-openai-apps-sdk/
22. Troubleshooting - OpenAI for developers, accessed November 25, 2025, https://developers.openai.com/apps-sdk/deploy/troubleshooting/
23. useChat - AI SDK UI, accessed November 25, 2025, https://ai-sdk.dev/docs/reference/ai-sdk-ui/use-chat
24. Builders Guide to the AI SDK | Vercel Academy, accessed November 25, 2025, https://vercel.com/academy/ai-sdk
25. How to build unified AI interfaces using the Vercel AI SDK - LogRocket Blog, accessed November 25, 2025, https://blog.logrocket.com/unified-ai-interfaces-vercel-sdk/
26. Chatbot Tool Usage - AI SDK UI, accessed November 25, 2025, https://ai-sdk.dev/docs/ai-sdk-ui/chatbot-tool-usage
27. Introducing Vercel AI SDK 3.2, accessed November 25, 2025, https://vercel.com/blog/introducing-vercel-ai-sdk-3-2
28. Generative UI without generating text - AI SDK - Vercel Community, accessed November 25, 2025, https://community.vercel.com/t/generative-ui-without-generating-text/6062
29. Chatbot Tool Usage - AI SDK UI, accessed November 25, 2025, https://sdk.vercel.ai/docs/ai-sdk-ui/chatbot-tool-usage
30. Using 4.1.46 with useChat and toolInvocation, after providing tool result there is no response from AI · Issue #5023 · vercel/ai - GitHub, accessed November 25, 2025, https://github.com/vercel/ai/issues/5023
31. Migrating from RSC to UI - AI SDK, accessed November 25, 2025, https://ai-sdk.dev/docs/ai-sdk-rsc/migrating-to-ui
32. Introducing AI SDK 3.0 with Generative UI support - Vercel, accessed November 25, 2025, https://vercel.com/blog/ai-sdk-3-generative-ui
33. Streaming React Components - AI SDK RSC, accessed November 25, 2025, https://ai-sdk.dev/docs/ai-sdk-rsc/streaming-react-components
34. Vercel AI SDK: Streaming React Components with Generative UI - Tim Avni, accessed November 25, 2025, https://www.timavni.com/blog/generative-ui-with-vercel-ai-sdk
35. How do I Stream OpenAI's completion API? - Stack Overflow, accessed November 25, 2025, https://stackoverflow.com/questions/73547502/how-do-i-stream-openais-completion-api
36. Stream OpenAI Chat Completions in JavaScript - Builder.io, accessed November 25, 2025, https://www.builder.io/blog/stream-ai-javascript
37. Bug: Azure OpenAI streaming responses include unexpected content_filter_offsets chunks ⟶ causes tool_calls parsing error in CopilotKit · Issue #2468 - GitHub, accessed November 25, 2025, https://github.com/CopilotKit/CopilotKit/issues/2468
38. Gemini OpenAI compatibility issue with tool_call + streaming - Google AI Developers Forum, accessed November 25, 2025, https://discuss.ai.google.dev/t/gemini-openai-compatibility-issue-with-tool-call-streaming/59886

39. Has anyone managed to get a tool_call working when stream=True? - API, accessed November 25, 2025, https://community.openai.com/t/has-anyone-managed-to-get-a-tool-call-working-when-stream-true/498867
40. ChatGPT error messages examples: Troubleshooting and solutions for 2025 - BytePlus, accessed November 25, 2025, https://www.byteplus.com/en/topic/546567
41. Building Smarter AI Apps with OpenAI and Vercel SDKs - Peerlist, accessed November 25, 2025, https://peerlist.io/jagss/articles/deep-dive-into-openai-and-vercel-ai-sdks--a-technical-perspe