Tutorial 31: React Vite ADK Integration - Custom UI with AG-UI Protocol

Difficulty: intermediate
Reading Time: 1.5 hours
Tags: ui, react, vite, ag-ui, custom-implementation, sse-streaming
Description: Build a fast, modern data analysis dashboard with Vite, React, TypeScript, and Google ADK using custom SSE streaming and AG-UI protocol.

:::info CUSTOM IMPLEMENTATION

This tutorial demonstrates a custom React frontend implementation using AG-UI protocol directly, WITHOUT CopilotKit.

Unlike Tutorial 30 which uses CopilotKit's pre-built components, this tutorial shows you how to build your own chat interface with manual SSE streaming, custom event handling, and tailored UX patterns like fixed sidebars for chart visualization.

Key Differences:
- ✅ Custom React components (no CopilotKit dependency)
- ✅ Manual SSE streaming with fetch() API
- ✅ Direct TOOL_CALL_RESULT event parsing
- ✅ Custom UI patterns (fixed sidebar, markdown rendering)
- ✅ Complete control over UX and styling

Refer to the [working implementation](https://github.com/raphaelmansuy/adk_training/tree/main/tutorial_implementation/tutorial31) (https://github.com/raphaelmansuy/adk_training/tree/main/tutorial_implementation/tutorial31) for complete, tested code.

:::

Tutorial 31: React Vite + ADK Integration (AG-UI Protocol)

Estimated Reading Time: 60-70 minutes
Difficulty Level: Intermediate
Prerequisites: Tutorial 29 (UI Integration Intro), Tutorial 30 (Next.js + ADK), Basic React knowledge

Table of Contents

1. Overview (#overview)
2. Why Vite for ADK Integration? (#why-vite-for-adk-integration)
3. Quick Start (5 Minutes) (#quick-start-5-minutes)
4. Building a Data Analysis Dashboard (#building-a-data-analysis-dashboard)
5. Advanced Features (#advanced-features)
6. Production Deployment (#production-deployment)
7. Vite vs Next.js Comparison (#vite-vs-nextjs-comparison)
8. Troubleshooting (#troubleshooting)
9. Next Steps (#next-steps)

Overview

What You'll Build

In this tutorial, you'll build a **real-time data analysis dashboard** using:

- **React 18** (with Vite) + **TypeScript**
- **Custom UI** (NO CopilotKit - manual SSE streaming)

- **AG-UI Protocol** (ag_ui_adk middleware)
- **Google ADK** (Agent backend with pandas tools)
- **Gemini 2.0 Flash Exp** (LLM)
- **Chart.js** + **react-chartjs-2** (Interactive visualizations)
- **react-markdown** (Rich text rendering with syntax highlighting)

Final Result:

```
┌──────────────────────────────────────────────────────────────┐
|  Data Analysis Dashboard                                     |
|  ├─ Upload CSV files                                         |
|  ├─ Ask questions about data ("What's the trend?")           |
|  ├─ Agent analyzes and generates insights                    |
|  ├─ Interactive charts render inline                         |
|  ├─ Export analysis reports                                  |
|  └─ Deploy to Netlify/Vercel in minutes                      |
└──────────────────────────────────────────────────────────────┘
```

Data Flow Architecture

```
User Uploads CSV → Agent Loads Data → User Asks Questions → Agent Analyzes → C
        ↓                ↓                ↓                ↓                ↓
   File Reader → load_csv_data() → SSE Stream → analyze_data() → TOOL_CALL_RES
   (Browser)     (Python Tool)     (AG-UI)      (Python Tool)     (Event Parsin
```

Tutorial Goals

✔️ Build custom React frontends without CopilotKit
✔️ Implement SSE streaming with fetch() API
✔️ Parse and handle AG-UI protocol events
✔️ Create a data analysis agent with pandas tools
✔️ Render charts from TOOL_CALL_RESULT events
✔️ Build fixed sidebar UI patterns for better UX
✔️ Handle file uploads and CSV processing
✔️ Deploy to production (Netlify + Cloud Run)

Why Vite for ADK Integration?

Vite Advantages

Feature	Benefit
⚡ **Instant Server Start**	Sub-second cold starts vs Next.js `3-5s`
🔥 **Lightning HMR**	Updates in <50ms, no page refresh
📦 **Optimized Build**	Smaller bundle sizes (50-70% of Next.js)
🎯 **Simple Config**	Single vite.config.js vs Next.js complexity
🚀 **Fast CI/CD**	2x-5x faster build times

When to Choose Vite

Choose Vite when you need:

- 🏃 Fast prototyping and development
- 📱 Single-page applications (SPAs)
- 🎨 Interactive dashboards and tools
- 💰 Smaller bundle sizes
- ⚙️ Simple deployment (static hosting)

Choose Next.js when you need:

- 🔍 SEO optimization (server-side rendering)
- 📄 Multi-page routing with App Router
- 🌐 Edge functions and middleware
- 📊 Complex server-side logic
- 🏢 Enterprise features (ISR, etc.)

End-to-End Data Flow

```
User Uploads CSV → Agent Loads Data → User Asks Questions → Agent Analyzes → C
        ↓                ↓                ↓                ↓                ↓
   File Reader → load_csv_data() → SSE Stream → analyze_data() → TOOL_CALL_RES
   (Browser)     (Python Tool)     (AG-UI)      (Python Tool)     (Event Parsin
```

Custom React + AG-UI Architecture

```
┌──────────────────────────────────────────────────────────────────────┐
│                      USER'S BROWSER                                  │
│  ┌──────────────────────────────────────────────────────────────┐   │
│  │  Vite Dev Server (Port 5173)                                 │   │
│  │  ├─ React 18 SPA (NO CopilotKit)                             │   │
│  │  ├─ Custom chat UI                                           │   │
│  │  ├─ Manual fetch() API calls                                 │   │
│  │  ├─ SSE streaming parser                                     │   │
│  │  ├─ Fixed sidebar for charts                                 │   │
│  │  └─ Hot Module Replacement (HMR)                             │   │
│  └──────────────────────────────────────────────────────────────┘   │
└──────────────────────────┬───────────────────────────────────────────┘
                           │
                           │ Direct HTTP + SSE
                           │ http://localhost:8000/api/copilotkit
                           │
┌──────────────────────────▼───────────────────────────────────────────┐
│              BACKEND SERVER (Port 8000)                              │
│  ┌──────────────────────────────────────────────────────────────┐   │
│  │  FastAPI + ag_ui_adk (AG-UI middleware)                      │   │
│  │  ├─ ADKAgent wrapping Agent                                  │   │
│  │  │  └─ Agent: gemini-2.0-flash-exp                           │   │
│  │  ├─ pandas tools (3 functions)                               │   │
│  │  │  ├─ load_csv_data                                         │   │
│  │  │  ├─ analyze_data                                          │   │
│  │  │  └─ create_chart → TOOL_CALL_RESULT                       │   │
│  │  └─ In-memory file storage (datasets dict)                   │   │
│  └──────────────────────┬───────────────────────────────────────┘   │
└─────────────────────────┴────────────────────────────────────────────┘
```

SSE Streaming Workflow

```
User Types Message
        ↓
   React onClick/sendMessage()
        ↓
   fetch('/api/copilotkit', {
     method: 'POST',
     body: JSON.stringify({messages, agent})
   })
        ↓
   Response.body.getReader() ← SSE Stream
        ↓
   Read chunks as they arrive
        ↓
   Split by '\n' (newline)
        ↓
   Parse 'data: {...}' lines
        ↓
   JSON.parse() each event
        ↓
   Handle Event Types:
   ├── TEXT_MESSAGE_CONTENT → Append to chat
   ├── TOOL_CALL_RESULT → Extract chart data
   └── Other events → Skip
        ↓
   Update React state → Re-render UI
```

Key Difference from Next.js:

- Vite uses **proxy configuration** instead of API routes
- Backend runs separately (same as Next.js pattern)
- Frontend is pure SPA (no server-side rendering)

Quick Start (5 Minutes)

Step 1: Create Vite Project

```
# Create Vite + React + TypeScript project
npm create vite@latest data-dashboard -- --template react-ts

cd data-dashboard

# Install visualization and markdown libraries
npm install chart.js react-chartjs-2
npm install react-markdown remark-gfm rehype-highlight rehype-raw
npm install highlight.js

npm install
```

Step 2: Configure Vite (Simple Config)

Update `vite.config.ts`:

```
// https://vitejs.dev/config/

  plugins: [react()],
  server: {
    port: 5173,
    // NO PROXY NEEDED - Direct connection to backend
        changeOrigin: true,
        rewrite: (path) => path.replace(/^\/api/, ""),
      },
    },
  },
});
```

What this does:

- Requests to `http://localhost:5173/api/copilotkit` → `http://localhost:8000/copilotkit`
- Avoids CORS issues during development
- Clean separation of concerns

Step 3: Create Data Analysis Agent

Create `agent/agent.py`:

```python
"""Data analysis ADK agent with pandas tools."""

import os
import io
import json
import pandas as pd
from typing import Dict, List, Any, Optional
from dotenv import load_dotenv
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
import uvicorn

# AG-UI ADK integration imports
from ag_ui_adk import ADKAgent, add_adk_fastapi_endpoint

# Google ADK imports
from google.adk.agents import Agent

load_dotenv()

# In-memory data storage (use Redis/DB in production)
uploaded_data = {}

def load_csv_data(file_name: str, csv_content: str) -> Dict[str, Any]:
    """
    Load CSV data into memory for analysis.

    Args:
        file_name: Name of the CSV file
        csv_content: CSV file content as string

    Returns:
        Dict with dataset info and preview
    """
    try:
        # Parse CSV
        df = pd.read_csv(io.StringIO(csv_content))

        # Store in memory
        uploaded_data[file_name] = df

        # Return summary
        return {
            "status": "success",
            "file_name": file_name,
            "rows": len(df),
```

```
            "columns": list(df.columns),
            "preview": df.head(5).to_dict(orient='records'),
            "dtypes": df.dtypes.astype(str).to_dict()
        }
    except Exception as e:
        return {
            "status": "error",
            "error": str(e)
        }

def analyze_data(
    file_name: str,
    analysis_type: str,
    columns: List[str] = None
) -> Dict[str, Any]:
    """
    Perform analysis on loaded dataset.

    Args:
        file_name: Name of dataset to analyze
        analysis_type: Type of analysis (summary, correlation, trend)
        columns: Optional list of columns to analyze

    Returns:
        Dict with analysis results
    """
    if file_name not in uploaded_data:
        return {"status": "error", "error": f"Dataset {file_name} not found"}

    df = uploaded_data[file_name]

    if columns:
        df = df[columns]

    results = {
        "status": "success",
        "file_name": file_name,
        "analysis_type": analysis_type
    }

    if analysis_type == "summary":
        results["data"] = {
            "describe": df.describe().to_dict(),
            "missing": df.isnull().sum().to_dict(),
            "unique": df.nunique().to_dict()
        }
```

```
    elif analysis_type == "correlation":
        # Only numeric columns
        numeric_df = df.select_dtypes(include=['number'])
        results["data"] = numeric_df.corr().to_dict()

    elif analysis_type == "trend":
        # Time series analysis
        if len(df) > 0:
            numeric_df = df.select_dtypes(include=['number'])
            results["data"] = {
                "mean": numeric_df.mean().to_dict(),
                "trend": "upward" if numeric_df.iloc[-1].sum() > numeric_df.il
            }

    return results

def create_chart(
    file_name: str,
    chart_type: str,
    x_column: str,
    y_column: str
) -> Dict[str, Any]:
    """
    Generate chart data for visualization.

    Args:
        file_name: Name of dataset
        chart_type: Type of chart (line, bar, scatter)
        x_column: Column for x-axis
        y_column: Column for y-axis

    Returns:
        Dict with chart configuration
    """
    if file_name not in uploaded_data:
        return {"status": "error", "error": f"Dataset {file_name} not found"}

    df = uploaded_data[file_name]

    if x_column not in df.columns or y_column not in df.columns:
        return {"status": "error", "error": "Invalid columns"}

    # Prepare chart data
    chart_data = {
        "status": "success",
        "chart_type": chart_type,
        "data": {
```

```
            "labels": df[x_column].tolist(),
            "values": df[y_column].tolist()
        },
        "options": {
            "x_label": x_column,
            "y_label": y_column,
            "title": f"{y_column} vs {x_column}"
        }
    }

    return chart_data

# Create ADK agent using the new API
adk_agent = Agent(
    name="data_analyst",
    model="gemini-2.5-flash",  # or "gemini-2.0-flash-exp"
    instruction="""You are a data analysis expert assistant.

Your capabilities:
- Load and analyze CSV datasets using load_csv_data()
- Perform statistical analysis using analyze_data()
- Generate insights and trends
- Create visualizations using create_chart()

Guidelines:
- Always start by loading data if not already loaded
- Explain your analysis clearly with markdown formatting
- Suggest relevant visualizations
- Highlight key insights with **bold** text
- Use statistical terms appropriately

When analyzing data:
1. Understand the dataset structure first
2. Perform appropriate analysis (summary, correlation, or trend)
3. Generate visualizations if helpful
4. Provide actionable insights

Be concise but thorough in your explanations.""",
    tools=[load_csv_data, analyze_data, create_chart]
)

# Wrap ADK agent with AG-UI middleware
agent = ADKAgent(
    adk_agent=adk_agent,
    app_name="data_analysis_app",
    user_id="demo_user",
    session_timeout_seconds=3600,
```

```
    use_in_memory_services=True
)

# Create FastAPI app
app = FastAPI(title="Data Analysis Agent API")

# Add CORS middleware for frontend
app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:5173", "http://localhost:3000"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Add ADK endpoint for CopilotKit
add_adk_fastapi_endpoint(app, agent, path="/api/copilotkit")

# Health check endpoint
@app.get("/health")
def health_check():
    """Health check endpoint."""
    return {
        "status": "healthy",
        "agent": "data_analyst",
        "datasets_loaded": list(uploaded_data.keys())
    }

# Run with: uvicorn agent:app --reload --port 8000
if __name__ == "__main__":
    port = int(os.getenv("PORT", "8000"))
    uvicorn.run(
        "agent:app",
        host="0.0.0.0",
        port=port,
        reload=True
    )
```

Create `agent/requirements.txt`:

```
google-genai>=1.15.0
fastapi>=0.115.0
uvicorn[standard]>=0.30.0
ag_ui_adk>=0.1.0
python-dotenv>=1.0.0
pandas>=2.0.0
```

Create `agent/.env`:

```
GOOGLE_API_KEY=your_gemini_api_key_here
```

Step 4: Create Custom React Frontend

File Upload and Processing Workflow

```
User Selects CSV File
        ↓
   React onChange Event
        ↓
   FileReader.readAsText()
        ↓
   File content loaded as string
        ↓
   sendMessage("Load this CSV file: " + content)
        ↓
   Manual fetch() to /api/copilotkit
        ↓
   Agent receives message with CSV data
        ↓
   Agent calls load_csv_data() tool
        ↓
   pandas reads CSV from string
        ↓
   Data stored in uploaded_data[file_name]
        ↓
   Agent confirms: "Data loaded successfully!"
        ↓
   User can now ask questions about the data
```

Update `src/App.tsx` with custom SSE streaming:

```
import './App.css'

interface Message {
  role: 'user' | 'assistant'
  content: string
}

interface ChartData {
  chart_type: 'line' | 'bar' | 'scatter'
  data: {
    labels: string[]
    values: number[]
  }
  options: {
    title: string
    x_label: string
    y_label: string
  }
}

function App() {
  const [messages, setMessages] = useState<Message[]>([])
  const [input, setInput] = useState('')
  const [isLoading, setIsLoading] = useState(false)
  const [currentChart, setCurrentChart] = useState<ChartData | null>(null)

  const handleFileUpload = async (event: React.ChangeEvent<HTMLInputElement>)
    const file = event.target.files?.[0]
    if (!file) return

    const reader = new FileReader()
    reader.onload = async (e) => {
      const content = e.target?.result as string

      // Send file to agent via manual SSE streaming
      await sendMessage(`Load this CSV file named "${file.name}":\n\n${content
    }
    reader.readAsText(file)
  }

  const sendMessage = async (messageContent: string) => {
    const userMessage: Message = { role: 'user', content: messageContent }
    setMessages(prev => [...prev, userMessage])
    setInput('')
    setIsLoading(true)
```

```
    try {
      // Manual fetch to AG-UI endpoint with SSE
      const response = await fetch('http://localhost:8000/api/copilotkit', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({
          messages: [...messages, userMessage],
          agent: 'data_analyst'
        })
      })

      const reader = response.body?.getReader()
      const decoder = new TextDecoder()
      let assistantMessage = ''

      while (true) {
        const { done, value } = await reader!.read()
        if (done) break

        const chunk = decoder.decode(value)
        const lines = chunk.split('\n')

        for (const line of lines) {
          if (line.startsWith('data: ')) {
            const jsonStr = line.slice(6)
            try {
              const jsonData = JSON.parse(jsonStr)

              // Handle different AG-UI event types
              if (jsonData.type === 'TEXT_MESSAGE_CONTENT') {
                assistantMessage += jsonData.content
                setMessages(prev => [
                  ...prev.slice(0, -1),
                  { role: 'assistant', content: assistantMessage }
                ])
              } else if (jsonData.type === 'TOOL_CALL_RESULT') {
                // Extract chart data from tool result
                const resultContent = typeof jsonData.content === 'string'
                  ? JSON.parse(jsonData.content)
                  : jsonData.content

                if (resultContent && resultContent.chart_type) {
                  setCurrentChart(resultContent)
                }
              }
            } catch (e) {
              // Skip invalid JSON
```

```
            }
          }
        }
      }
    } catch (error) {
      console.error('Error:', error)
      setMessages(prev => [
        ...prev,
        { role: 'assistant', content: 'Error: Could not get response from serv
      ])
    } finally {
      setIsLoading(false)
    }
  }

  return (
    <div className="app-container">
      <div className="dashboard">
        <header className="header">
          <h1>📊 Data Analysis Dashboard</h1>
          <p>Upload CSV data and ask questions to get insights</p>
        </header>

        {/* File Upload */}
        <div className="upload-section">
          <label htmlFor="file-upload" className="upload-button">
            📁 Drop CSV files here or browse
          </label>
          <input
            id="file-upload"
            type="file"
            accept=".csv"
            onChange={handleFileUpload}
            style={{ display: 'none' }}
          />
        </div>

        {/* Custom Chat Interface */}
        <div className="chat-container">
          {messages.map((msg, i) => (
            <div key={i} className={`message ${msg.role}`}>
              <ReactMarkdown>{msg.content}</ReactMarkdown>
            </div>
          ))}
          {isLoading && <div className="loading">Thinking...</div>}
        </div>
```

```
          {/* Input */}
          <div className="input-container">
            <input
              value={input}
              onChange={(e) => setInput(e.target.value)}
              onKeyPress={(e) => e.key === 'Enter' && sendMessage(input)}
              placeholder="Ask about your data..."
              disabled={isLoading}
            />
            <button onClick={() => sendMessage(input)} disabled={isLoading}>
              Send
            </button>
          </div>
        </div>

        {/* Fixed Sidebar for Charts */}
        {currentChart && (
          <aside className="chart-sidebar">
            <button onClick={() => setCurrentChart(null)}>x</button>
            {currentChart.chart_type === 'line' && (
              <Line data={/* format chart data */} />
            )}
            {currentChart.chart_type === 'bar' && (
              <Bar data={/* format chart data */} />
            )}
            {currentChart.chart_type === 'scatter' && (
              <Scatter data={/* format chart data */} />
            )}
          </aside>
        )}
      </div>
    )
  }
```

Update `src/App.css`:

```
.app-container {
  min-height: 100vh;
  background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
  padding: 2rem;
}

.dashboard {
  max-width: 1200px;
  margin: 0 auto;
}

.header {
  text-align: center;
  color: white;
  margin-bottom: 2rem;
}

.header h1 {
  font-size: 3rem;
  margin-bottom: 0.5rem;
}

.header p {
  font-size: 1.2rem;
  opacity: 0.9;
}

.upload-section {
  background: white;
  padding: 2rem;
  border-radius: 12px;
  margin-bottom: 2rem;
  text-align: center;
  box-shadow: 0 10px 30px rgba(0, 0, 0, 0.2);
}

.upload-button {
  display: inline-block;
  padding: 1rem 2rem;
  background: #667eea;
  color: white;
  border-radius: 8px;
  cursor: pointer;
  font-weight: 600;
  transition: all 0.3s ease;
}
```

```
.upload-button:hover {
  background: #764ba2;
  transform: translateY(-2px);
  box-shadow: 0 5px 15px rgba(0, 0, 0, 0.3);
}

.file-name {
  margin-left: 1rem;
  color: #28a745;
  font-weight: 600;
}

.chat-container {
  background: white;
  border-radius: 12px;
  overflow: hidden;
  box-shadow: 0 10px 30px rgba(0, 0, 0, 0.2);
  height: 600px;
}
```

Step 5: Run Everything

```
# Terminal 1: Run agent
cd agent
python -m venv venv
source venv/bin/activate  # Windows: venv\Scripts\activate
pip install -r requirements.txt
python agent.py

# Terminal 2: Run Vite frontend
cd ..
npm run dev
```

Open http://localhost:5173 - Your data analysis dashboard is live! 🎉

Try it:

1. Upload a CSV file (sales data, etc.)
2. Ask: "What are the key statistics?"
3. Ask: "Show me a chart of sales over time"
4. Watch the agent analyze and visualize your data!

Building a Data Analysis Dashboard

Let's enhance our dashboard with real data visualization.

Feature 1: Interactive Charts

Install Chart.js:

```
npm install chart.js react-chartjs-2
```

Create `src/components/ChartRenderer.tsx`:

```
import {
  Chart as ChartJS,
  CategoryScale,
  LinearScale,
  PointElement,
  LineElement,
  BarElement,
  Title,
  Tooltip,
  Legend,
} from 'chart.js'

// Register Chart.js components
ChartJS.register(
  CategoryScale,
  LinearScale,
  PointElement,
  LineElement,
  BarElement,
  Title,
  Tooltip,
  Legend
)

interface ChartData {
  chart_type: string
  data: {
    labels: string[]
    values: number[]
  }
  options: {
    x_label: string
    y_label: string
    title: string
  }
}

interface ChartRendererProps {
  chartData: ChartData
}

  const data = {
    labels: chartData.data.labels,
    datasets: [
      {
        label: chartData.options.y_label,
```

```
        data: chartData.data.values,
        backgroundColor: 'rgba(102, 126, 234, 0.5)',
        borderColor: 'rgba(102, 126, 234, 1)',
        borderWidth: 2,
      },
    ],
  }

  const options = {
    responsive: true,
    plugins: {
      legend: {
        position: 'top' as const,
      },
      title: {
        display: true,
        text: chartData.options.title,
      },
    },
    scales: {
      x: {
        title: {
          display: true,
          text: chartData.options.x_label,
        },
      },
      y: {
        title: {
          display: true,
          text: chartData.options.y_label,
        },
      },
    },
  }

  // Render appropriate chart type
  switch (chartData.chart_type) {
    case 'line':
      return <Line data={data} options={options} />
    case 'bar':
      return <Bar data={data} options={options} />
    case 'scatter':
      return <Scatter data={data} options={options} />
    default:
      return <div>Unsupported chart type: {chartData.chart_type}</div>
  }
}
```

Feature 2: Chart Rendering from TOOL_CALL_RESULT Events

The custom implementation extracts chart data from AG-UI protocol events:

```
// In the SSE streaming loop (from App.tsx)
for (const line of lines) {
  if (line.startsWith('data: ')) {
    const jsonStr = line.slice(6)
    try {
      const jsonData = JSON.parse(jsonStr)

      // Extract chart data from TOOL_CALL_RESULT events
      if (jsonData.type === 'TOOL_CALL_RESULT') {
        const resultContent = typeof jsonData.content === 'string'
          ? JSON.parse(jsonData.content)
          : jsonData.content

        // Check if this is chart data
        if (resultContent && resultContent.chart_type) {
          setCurrentChart(resultContent)
        }
      }
    } catch (e) {
      // Skip invalid JSON
    }
  }
}
```

TOOL_CALL_RESULT Processing Flow

```
Agent Decides to Create Chart
        ↓
   Calls create_chart() tool
        ↓
   Tool returns chart config:
   {
     "status": "success",
     "chart_type": "line",
     "data": {"labels": [...], "values": [...]},
     "options": {"title": "...", "x_label": "..."}
   }
        ↓
   AG-UI wraps in TOOL_CALL_RESULT event
        ↓
   SSE stream sends: data: {
     "type": "TOOL_CALL_RESULT",
     "content": "{chart config JSON}"
   }
        ↓
   Frontend parses event
        ↓
   Extracts chart data from content
        ↓
   setCurrentChart(chartData) → React state
        ↓
   Fixed sidebar re-renders with Chart.js
        ↓
   User sees interactive visualization
```

Key Points:
- Agent calls `create_chart()` tool
- Backend returns chart data via `TOOL_CALL_RESULT` event
- Frontend extracts and stores chart data in state
- Chart renders in fixed sidebar with Chart.js components
- No generative UI framework needed - direct state management! 📊

Feature 3: Data Table View

Create `src/components/DataTable.tsx`:

```
interface DataTableProps {
  data: Array<Record<string, any>>
  columns: string[]
}

  return (
    <div className="data-table-container">
      <table className="data-table">
        <thead>
          <tr>
            {columns.map((col) => (
              <th key={col}>{col}</th>
            ))}
          </tr>
        </thead>
        <tbody>
          {data.map((row, idx) => (
            <tr key={idx}>
              {columns.map((col) => (
                <td key={col}>{row[col]}</td>
              ))}
            </tr>
          ))}
        </tbody>
      </table>
    </div>
  )
}
```

Add CSS in `src/App.css`:

```
.data-table-container {
  max-height: 400px;
  overflow: auto;
  margin: 1rem 0;
  border-radius: 8px;
  border: 1px solid #e0e0e0;
}

.data-table {
  width: 100%;
  border-collapse: collapse;
}

.data-table thead {
  background: #667eea;
  color: white;
  position: sticky;
  top: 0;
}

.data-table th,
.data-table td {
  padding: 0.75rem;
  text-align: left;
  border-bottom: 1px solid #e0e0e0;
}

.data-table tbody tr:hover {
  background: #f5f5f5;
}
```

Feature 4: Export Analysis Report

Add export functionality:

```
const exportAnalysis = () => {
  // Collect all analysis results
  const report = {
    timestamp: new Date().toISOString(),
    file: uploadedFile,
    analysis: "... collected from agent responses ...",
    charts: "... chart configurations ..."
  }

  // Convert to JSON
  const blob = new Blob([JSON.stringify(report, null, 2)], {
    type: 'application/json'
  })

  // Download
  const url = URL.createObjectURL(blob)
  const a = document.createElement('a')
  a.href = url
  a.download = `analysis_${Date.now()}.json`
  a.click()
  URL.revokeObjectURL(url)
}

// Add button to UI
<button onClick={exportAnalysis} className="export-button">
  📥 Export Report
</button>
```

Advanced Features

Feature 1: Real-Time Collaboration

Share dashboard state with the agent:

```
function App() {
  const [sharedState, setSharedState] = useState({
    uploadedFiles: [],
    currentAnalysis: null,
    activeDataset: null,
  });

  // Include state in messages for agent context
  const sendMessageWithContext = async (userMessage: string) => {
    const contextMessage = {
      role: 'system',
      content: `Current state: ${JSON.stringify(sharedState)}`
    }

    const response = await fetch('http://localhost:8000/api/copilotkit', {
      method: 'POST',
      body: JSON.stringify({
        messages: [contextMessage, ...messages, { role: 'user', content: userM
        agent: 'data_analyst'
      })
    })
    // ... handle response
  }
}
```

No special hooks needed - just include state in message history!

Feature 2: Analysis History Persistence

Persist analysis history with localStorage:

```
const [analysisHistory, setAnalysisHistory] = useState<Analysis[]>(() => {
  // Load from localStorage on mount
  const saved = localStorage.getItem('analysis_history')
  return saved ? JSON.parse(saved) : []
});

// Save to localStorage whenever history changes
useEffect(() => {
  localStorage.setItem('analysis_history', JSON.stringify(analysisHistory))
}, [analysisHistory])

// Add analysis to history
const saveAnalysis = (analysis: Analysis) => {
  setAnalysisHistory((prev) => [...prev, analysis])
}

// Agent doesn't need special hooks - just include history in messages:
const messagesWithHistory = [
  {
    role: 'system',
    content: `Previous analyses: ${JSON.stringify(analysisHistory)}`
  },
  ...messages
]
```

Key Difference: No special agent memory framework needed - use standard React patterns!

Feature 3: Multi-File Analysis

Compare multiple datasets:

```python
# In agent.py
def compare_datasets(
    file_names: List[str],
    metric: str
) -> Dict[str, Any]:
    """Compare metric across multiple datasets."""
    comparison = {}

    for name in file_names:
        if name in uploaded_data:
            df = uploaded_data[name]
            if metric in df.columns:
                comparison[name] = df[metric].mean()

    return {
        "status": "success",
        "comparison": comparison,
        "winner": max(comparison, key=comparison.get) if comparison else None
    }
```

Production Deployment

Deployment Architecture Comparison

Development Setup:

```
Browser (5173) ←── Proxy ──→ FastAPI (8000)
     ↓                          ↓
   Vite Dev                  ADK Agent
   Server                    + AG-UI
```

Production Setup:

```
Browser ←── HTTPS ──→ Netlify/Vercel ←── HTTPS ──→ Cloud Run
                            ↓                          ↓
                        Static Files              ADK Agent
                                                   + AG-UI
```

Option 1: Deploy to Netlify

Deployment Workflow:

```
Local Development
        ↓
   npm run build          (Create dist/ folder)
        ↓
   gcloud run deploy      (Deploy agent to Cloud Run)
        ↓
   Update API_URL         (Point to Cloud Run URL)
        ↓
   netlify deploy         (Upload static files)
        ↓
   Configure CORS         (Allow Netlify domain)
        ↓
   Test live app          (End-to-end verification)
```

Step 1: Build Frontend

```
# Create production build
npm run build

# Output in dist/ directory
```

Step 2: Deploy Agent to Cloud Run

```
# Deploy agent (same as Tutorial 30)
cd agent
gcloud run deploy data-analysis-agent \
  --source=. \
  --region=us-central1 \
  --allow-unauthenticated \
  --set-env-vars="GOOGLE_API_KEY=your_key"

# Get URL: https://data-analysis-agent-xyz.run.app
```

Step 3: Update Frontend for Production

Create `src/config.ts`:

```
  ? "https://data-analysis-agent-xyz.run.app"
  : "http://localhost:8000";
```

Update `src/App.tsx`:

```
// Use in fetch calls
const response = await fetch(`${API_URL}/api/copilotkit`, {
  method: 'POST',
  // ... rest of config
})
```

Step 4: Deploy to Netlify

```
# Install Netlify CLI
npm install -g netlify-cli

# Login
netlify login

# Deploy
netlify deploy --prod --dir=dist

# Or connect GitHub repo for auto-deploy
netlify init
```

Done! Your app is live at `https://data-dashboard-xyz.netlify.app` 🚀

Option 2: Deploy to Vercel

```
# Install Vercel CLI
npm install -g vercel

# Deploy
vercel

# Set environment variable
vercel env add VITE_API_URL production
# Enter: https://data-analysis-agent-xyz.run.app

# Redeploy with env
vercel --prod
```

Done! Your app is live at `https://data-dashboard.vercel.app` 🎉

Vite vs Next.js Comparison

Development Experience

Aspect	Vite	Next.js 15
Cold Start	<1s	3-5s
HMR Speed	<50ms	200-500ms
Build Time	10-30s	30-120s
Bundle Size	100-200KB	200-400KB
Config	Simple	Complex

Feature Comparison

Feature	Vite	Next.js 15
SPA Support	✅ Native	✅ Via export
SSR	⚠️ Manual (Vite SSR)	✅ Built-in
API Routes	❌ Proxy only	✅ Full support
File Routing	❌ Manual	✅ Built-in
Image Optimization	❌ Manual	✅ Built-in
Middleware	❌ None	✅ Edge runtime
Static Export	✅ Native	✅ Built-in
Hot Reload	✅ Lightning fast	✅ Good

When to Use Each

Use Vite for:

- ⚡ Prototypes and MVPs
- 🎨 Dashboards and admin panels
- 📊 Data visualization tools
- 🔧 Internal tools
- 📱 SPAs without SEO needs
- 🚀 Fast iteration needed

Use Next.js for:

- 🔍 SEO-critical sites
- 📄 Multi-page websites
- 🌐 Public-facing apps
- 🏢 Enterprise applications
- 📊 Complex routing needs
- 🔐 Server-side auth

Code Comparison

Vite + Custom React (Tutorial 31):

```
### Code Comparison

**Vite + Custom React** (Tutorial 31):

```typescript
// Single App.tsx file with full control
// Manual SSE streaming with fetch()
// Custom UI components
// Direct state management
// ~200 lines of code for complete chat interface

const response = await fetch('http://localhost:8000/api/copilotkit', {
  method: 'POST',
  body: JSON.stringify({ messages, agent: 'data_analyst' })
})
// Parse SSE events manually, extract TOOL_CALL_RESULT, render charts
```
```

Next.js + CopilotKit (Tutorial 30):

```
// app/layout.tsx - CopilotKit wrapper
// app/page.tsx - Main page with <CopilotChat />
// app/api/copilotkit/route.ts - API route handler

// Pre-built components, less code, standard UX, faster to build

<CopilotKit runtimeUrl="/api/copilotkit">
  <CopilotChat /> {/* ~10 lines for basic chat */}
</CopilotKit>
```

Implementation Comparison Diagram

```
Feature Category          Vite + Custom React          Next.js + CopilotKit
──────────────────────────────────────────────────────────────────────────
Code Volume               High (200+ lines)            Low (10-50 lines)
UI Control                Full control                 Limited to CopilotKit
UX Flexibility            Custom (fixed sidebar!)      Standard chat UI
Learning Curve            Higher (manual streaming)    Lower (pre-built)
Bundle Size               Smaller (no framework)       Larger (framework)
Development Speed         Slower initial              Faster initial
Maintenance               More complex                 Simpler
Customization             Unlimited                    Limited
Performance               Better (no framework)        Good
Deployment                Static hosting               Server required
```

```
**Next.js + CopilotKit** (Tutorial 30):

```typescript
// app/layout.tsx - CopilotKit wrapper
// app/page.tsx - Main page with <CopilotChat />
// app/api/copilotkit/route.ts - API route handler

// Pre-built components, less code, less control

<CopilotKit runtimeUrl="/api/copilotkit">
  <CopilotChat />  {/* ~10 lines for basic chat */}
</CopilotKit>
```
```

Trade-offs:

- Custom React: More code, full control, custom UX (fixed sidebar!)
- CopilotKit: Less code, standard UX, faster to build

Troubleshooting

SSE Streaming Debug Flow

```
SSE Not Working?
        ↓
   Check browser console for errors
        ↓
   Is fetch() getting HTTP 200?
        ├── YES → Check response.body exists
        └── NO → Check backend running on port 8000
        ↓
   Is reader getting chunks?
        ├── YES → Check 'data: ' lines parsing
        └── NO → Check fetch URL and method
        ↓
   Are events being parsed?
        ├── YES → Check event.type handling
        └── NO → Check JSON.parse() not failing
        ↓
   Is UI updating?
        ├── YES → Success!
        └── NO → Check React state updates
```

Issue 1: SSE Streaming Not Working

Symptoms:

- No response from agent
- Messages appear to send but no reply
- Browser console shows no errors

Solution:

```
// Check fetch() is configured correctly
const response = await fetch('http://localhost:8000/api/copilotkit', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    messages: [...messages, userMessage],
    agent: 'data_analyst'  // CRITICAL: Must match agent name in backend
  })
})

// Verify response is readable stream
if (!response.body) {
  console.error('Response body is null - check backend')
  return
}

// Check for response errors
if (!response.ok) {
  console.error(`HTTP ${response.status}: ${response.statusText}`)
  const text = await response.text()
  console.error('Response:', text)
  return
}
```

Debug steps:

1. Check backend is running: `curl http://localhost:8000/health`
2. Verify agent name matches: Check `agent/agent.py` for `name="data_analyst"`
3. Open browser DevTools → Network tab → Check `/api/copilotkit` request
4. Look for backend errors in terminal running `make dev-agent`

Issue 2: CORS in Production

Symptoms:

- Works locally, fails in production
- CORS errors in browser console

Solution:

```python
# agent/agent.py - Update CORS for production
from fastapi.middleware.cors import CORSMiddleware

app.add_middleware(
    CORSMiddleware,
    allow_origins=[
        "https://data-dashboard.netlify.app",  # Your Netlify domain
        "https://data-dashboard.vercel.app",   # Your Vercel domain
        "http://localhost:5173",  # Local dev
    ],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Issue 3: Large File Upload Issues

Symptoms:

- Upload fails for files >1MB
- Timeout errors

Solution:

```python
# agent/agent.py - Increase limits
from fastapi import FastAPI, File, UploadFile

app = FastAPI()

# Increase body size limit
@app.post("/upload")
async def upload_file(file: UploadFile = File(...)):
    # Handle large files
    content = await file.read()
    return {"size": len(content)}

# In uvicorn startup
uvicorn.run(
    app,
    host="0.0.0.0",
    port=8000,
    limit_concurrency=100,
    limit_max_requests=1000,
    timeout_keep_alive=30
)
```

Issue 4: TOOL_CALL_RESULT Event Not Parsed

Symptoms:

- Agent responds but charts don't appear
- Console shows "Cannot read property 'chart_type' of undefined"

Solution:

```
// Proper TOOL_CALL_RESULT parsing
if (jsonData.type === 'TOOL_CALL_RESULT') {
  // Content might be string or object
  const resultContent = typeof jsonData.content === 'string'
    ? JSON.parse(jsonData.content)  // Parse if string
    : jsonData.content               // Use directly if object

  // Validate chart data structure
  if (resultContent &&
      resultContent.chart_type &&
      resultContent.data &&
      resultContent.data.labels &&
      resultContent.data.values) {
    console.log('Valid chart data:', resultContent)
    setCurrentChart(resultContent)
  } else {
    console.warn('Invalid chart data structure:', resultContent)
  }
}
```

Debug checklist:

1. Check backend `create_chart` returns correct format
2. Verify `status: "success"` in tool result
3. Ensure `chart_type` is 'line', 'bar', or 'scatter'
4. Confirm arrays: `data.labels` (strings), `data.values` (numbers)

Issue 5: Chart.js Not Registered

Symptoms:

- Error: "category is not a registered scale"
- Charts show blank canvas

Solution:

```
// Import and register ALL Chart.js components at app startup
import {
  Chart as ChartJS,
  CategoryScale,
  LinearScale,
  PointElement,
  LineElement,
  BarElement,
  Title,
  Tooltip,
  Legend,
} from "chart.js";

// Register ONCE at app initialization (top of App.tsx)
ChartJS.register(
  CategoryScale,
  LinearScale,
  PointElement,
  LineElement,
  BarElement,
  Title,
  Tooltip,
  Legend,
);
```

Next Steps

You've Mastered Vite + ADK! 🎉

You now know how to:

✔️ Build lightning-fast React + Vite + ADK apps
✔️ Create data analysis dashboards
✔️ Implement generative UI with Chart.js
✔️ Handle file uploads and processing
✔️ Deploy to Netlify or Vercel
✔️ Compare Vite vs Next.js approaches

Continue Learning

Tutorial 32: Streamlit + ADK Integration
Build data apps with pure Python (no frontend code!)

Tutorial 33: Slack Bot Integration
Create team collaboration bots for Slack

Tutorial 35: AG-UI Deep Dive
Master advanced features: multi-agent UI, custom protocols

Additional Resources

- Vite Documentation (https://vitejs.dev/)
- CopilotKit + Vite Guide (https://docs.copilotkit.ai/guides/vite)
- Chart.js Documentation (https://www.chartjs.org/)
- Example: gemini-fullstack (https://github.com/google/adk-samples/tree/main/gemini-fullstack)

🎉 **Tutorial 31 Complete!**

Next: Tutorial 32: Streamlit + ADK Integration (./32_streamlit_adk_integration.md)

Questions or feedback? Open an issue on the ADK Training Repository (https://github.com/google/adk-training).

Generated on 2025-10-21 09:03:04 from 31_react_vite_adk_integration.md

Source: Google ADK Training Hub