Document

Tutorial 29: Introduction to UI Integration & AG-UI Protocol

:::tip Working Implementation Available

A complete, tested implementation is available in the repository!

👉 View Implementation (./../../tutorial_implementation/tutorial29)

The implementation includes:
- ✔️ Python ADK agent with AG-UI protocol integration
- ✔️ FastAPI backend with middleware for CopilotKit compatibility
- ✔️ React + Vite frontend with custom UI (no CopilotKit components)
- ✔️ Tailwind CSS for modern styling
- ✔️ Comprehensive test suite (15+ tests passing)
- ✔️ Complete documentation and Makefile with dev commands

Implementation Note: The tutorial29 implementation uses a **custom React UI** with direct API calls instead of CopilotKit components. This demonstrates the underlying AG-UI Protocol and gives you full control over the UI. For production apps with pre-built components, see Tutorial 30 (Next.js with CopilotKit).

Quick Start:

```
cd tutorial_implementation/tutorial29
make setup
# Configure your API key in agent/.env
make dev
# Open http://localhost:5173
```

:::

:::info Verify Runner API Usage

CRITICAL: ADK v1.16+ changed the Runner API. All code examples use the correct pattern.

Correct Runner API (verified in source code):
- ✔️ CORRECT: `from google.adk.runners import InMemoryRunner`
- ✔️ CORRECT: `runner = InMemoryRunner(agent=agent, app_name='app')`
- ✔️ CORRECT: Create session, then use `async for event in runner.run_async(...)`

Common Mistakes to Avoid:
- ❌ WRONG: `from google.adk.agents import Runner` - doesn't exist in v1.16+
- ❌ WRONG: `runner = Runner()` - use InMemoryRunner
- ❌ WRONG: `await runner.run_async(query, agent=agent)` - use async iteration

Source: `/research/adk-python/src/google/adk/runners.py`

:::

Estimated Reading Time: 35-45 minutes
Difficulty Level: Intermediate
Prerequisites: Tutorials 1-3 (ADK Basics), Tutorial 14 (Streaming & SSE)

Table of Contents

Overview

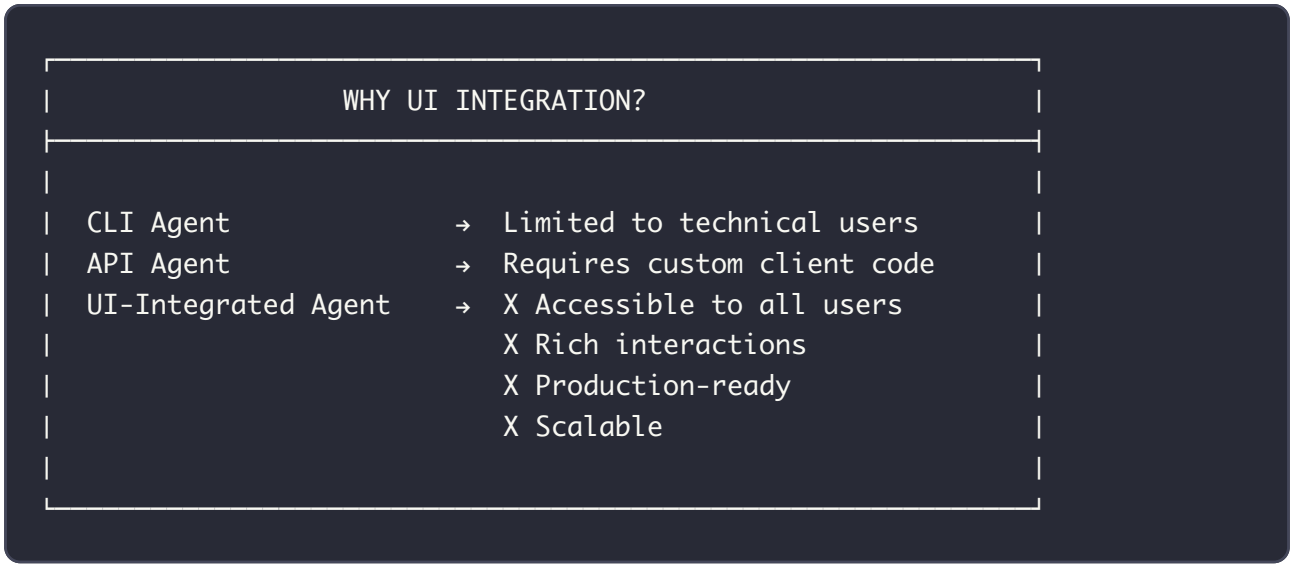
What You'll Learn

In this tutorial, you'll master the fundamentals of integrating Google ADK agents with user interfaces. You'll understand:

- **The UI integration landscape** - Different approaches and when to use each
- **AG-UI Protocol** - The official protocol for agent-UI communication
- **Integration patterns** - React/Next.js, Streamlit, Slack, and event-driven architectures
- **Decision framework** - How to choose the right approach for your use case
- **Architecture patterns** - Production-ready deployment strategies

Why UI Integration Matters

While ADK agents are powerful on their own, connecting them to user interfaces unlocks their full potential:



Real-World Use Cases:

- **Customer Support Chatbots** - Web-based chat interfaces for customer service
- **Data Analysis Dashboards** - Interactive ML/AI tools for business intelligence
- **Team Collaboration Bots** - Slack/Teams bots for enterprise workflows
- **Document Processing Systems** - Event-driven UI for document pipelines

The ADK UI Integration Landscape

Overview of Integration Options

Google ADK supports multiple UI integration paths, each optimized for different use cases:

```
┌─────────────────────────────────────────────────────────────┐
│                ADK UI INTEGRATION OPTIONS                   │
├─────────────────────────────────────────────────────────────┤
│                                                             │
│  1. AG-UI Protocol (CopilotKit)                             │
│     ├─ Best for: React/Next.js web applications             │
│     ├─ Features: Pre-built components, TypeScript SDK       │
│     └─ Tutorials: 29, 30, 31, 35                            │
│                                                             │
│  2. Native ADK API (HTTP/SSE/WebSocket)                     │
│     ├─ Best for: Custom implementations, any framework      │
│     ├─ Features: Full control, no dependencies              │
│     └─ Tutorials: 14, 29, 32                                │
│                                                             │
│  3. Direct Python Integration                               │
│     ├─ Best for: Data apps, Streamlit, internal tools       │
│     ├─ Features: In-process, no HTTP overhead               │
│     └─ Tutorial: 32                                         │
│                                                             │
│  4. Messaging Platform Integration                          │
│     ├─ Best for: Team collaboration, Slack/Teams bots       │
│     ├─ Features: Native platform UX, rich formatting        │
│     └─ Tutorial: 33                                         │
│                                                             │
│  5. Event-Driven Architecture                               │
│     ├─ Best for: High-scale, asynchronous processing        │
│     ├─ Features: Pub/Sub, scalable, decoupled               │
│     └─ Tutorial: 34                                         │
│                                                             │
└─────────────────────────────────────────────────────────────┘
```

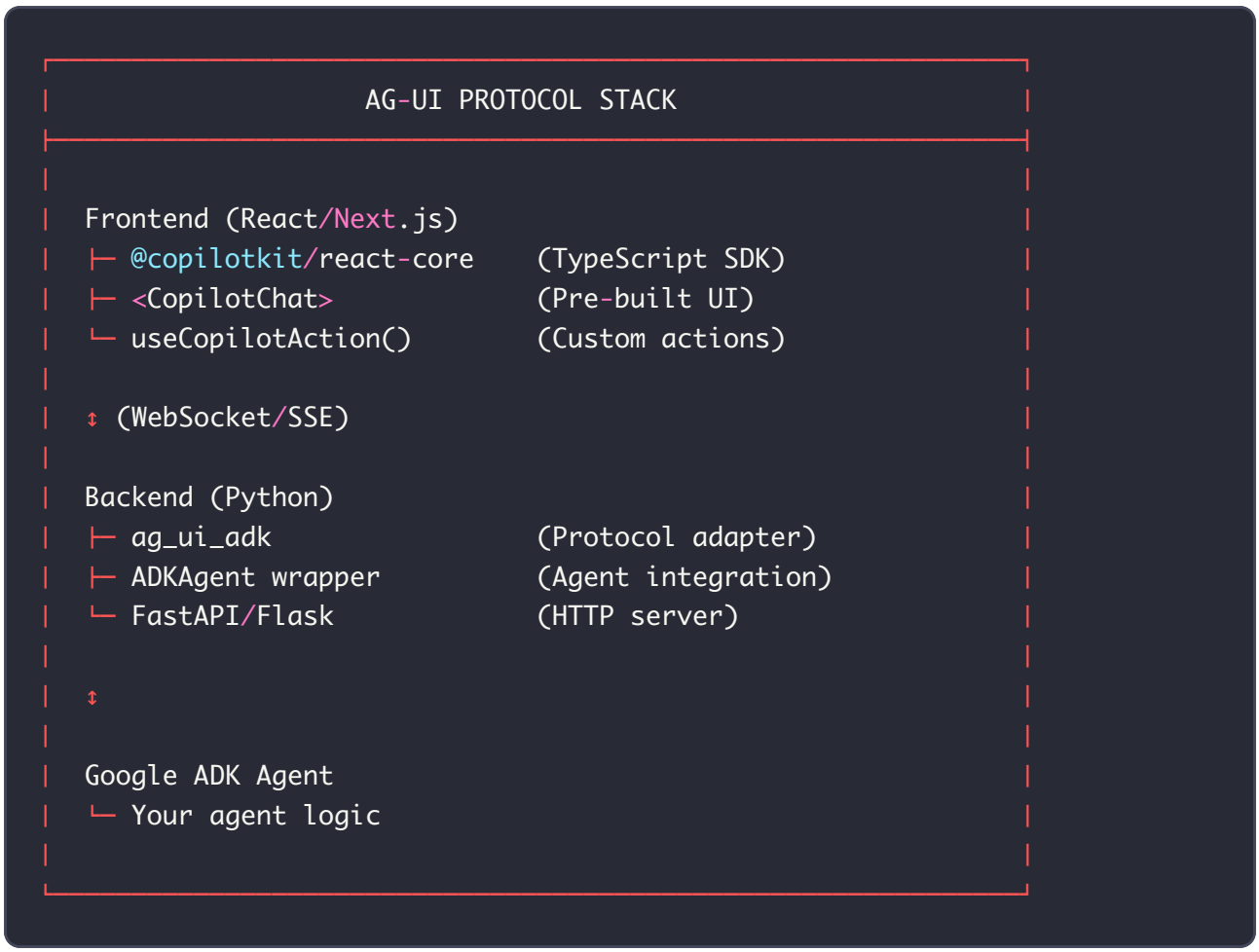
Comparison Matrix

Approach	Best For	Complexity	Scalability	Time to Production
AG-UI Protocol	Modern web apps	Low	High	⚡ Fast (hours)
Native API	Custom frameworks	Medium	High	🔨 Moderate (days)
Direct Python	Data apps	Low	Medium	⚡ Fast (hours)
Slack/Teams	Team tools	Low	High	⚡ Fast (hours)
Pub/Sub	Event-driven	High	Very High	🔨 Complex (weeks)

Understanding the AG-UI Protocol

What is AG-UI?

AG-UI (Agent-Generative UI) is an open protocol for agent-user interaction, developed through an **official partnership between Google ADK and CopilotKit**. It provides a standardized way for AI agents to communicate with web UIs.



Key Features

1. Event-Based Communication

AG-UI uses events for agent-UI communication:

Example event messages:

```
// Frontend sends action request
{
  "type": "action",
  "name": "analyze_data",
  "arguments": { "dataset": "sales_2024.csv" }
}

// Agent sends progress updates
{
  "type": "textMessage",
  "content": "Analyzing sales data..."
}

// Agent sends result
{
  "type": "actionResult",
  "actionName": "analyze_data",
  "result": { "revenue": 1500000, "growth": 0.15 }
}
```

2. Pre-Built React Components

```
// Drop-in chat UI with zero configuration
<CopilotChat />;
```

3. Generative UI

Agents can render custom React components:

```
# Agent returns structured data
return {
    "component": "DataVisualization",
    "props": {
        "chartType": "bar",
        "data": sales_data
    }
}
```

4. Production-Ready Middleware

```
from ag_ui_adk import ADKAgent
from google.adk.agents import Agent

# Create ADK agent and wrap it
adk_agent = Agent(
    name="customer_support",
    model="gemini-2.0-flash-exp"
)
agent = ADKAgent(adk_agent=adk_agent, app_name="customer_support")
```

Why AG-UI Protocol?

✔️ **Advantages**:

- **Official Support** - Partnership with Google ADK team
- **Pre-Built Components** - `<CopilotChat>`, `<CopilotTextarea>`
- **TypeScript SDK** - Type-safe React integration
- **Extensive Examples** - Production-ready code
- **Active Community** - Discord, GitHub discussions
- **Comprehensive Testing** - 271 tests passing

⚠️ **Considerations**:

- Additional dependency (CopilotKit packages)
- TypeScript-first ecosystem (though JS works)
- Event translation overhead (minimal, ~5ms)

Integration Approaches

Approach 1: AG-UI Protocol (Recommended for Web Apps)

When to Use:

- Building React/Next.js web applications
- Need pre-built UI components

- Want TypeScript type safety
- Prefer official, well-documented patterns

Architecture:

```
┌─────────────────────────────────────────────────────────────┐
│                                                             │
│  User Browser                                               │
│  ├─ React App                                               │
│  ├─ CopilotKit Provider                                     │
│  └─ <CopilotChat> component                                 │
│                                                             │
│         ↕ (WebSocket/SSE)                                   │
│                                                             │
│  Backend Server (FastAPI)                                   │
│  ├─ ag_ui_adk                  (AG-UI Protocol adapter)     │
│  ├─ ADKAgent wrapper           (Session management)         │
│  └─ Your ADK agent             (google.adk.agents.LlmAgent) │
│                                                             │
│         ↕                                                   │
│                                                             │
│  Gemini API                                                 │
│                                                             │
└─────────────────────────────────────────────────────────────┘
```

Complete Message Flow:


END-TO-END MESSAGE FLOW
Step 1: User Input
User types: "What is ADK?"
Frontend captures input
Step 2: Frontend Processing
- Create message object: {role: "user", content: "..."}
- Add to local state (immediate UI update)
- Prepare API request with session context
Step 3: HTTP/WebSocket Request
POST /api/copilotkit
{
threadId: "session-123",
messages: [{role: "user", content: "What is ADK?"}]
}
Step 4: Backend Processing
ag_ui_adk receives request
- Validates session
- Retrieves conversation history
- Converts AG-UI format to ADK format
Step 5: Agent Execution
ADK Agent processes request
- Constructs prompt with context
- Calls Gemini API
- Streams response tokens

```
  |                          v                                          |
  |  Step 6: Response Streaming                                         |
  |  +-------------------------------------------------------------+    |
  |  | Backend streams events:                                     |    |
  |  | Event 1: {type: "TEXT_MESSAGE", delta: "ADK is..."}         |    |
  |  | Event 2: {type: "TEXT_MESSAGE", delta: "a framework"}       |    |
  |  | Event 3: {type: "TEXT_MESSAGE", delta: "for..."}            |    |
  |  | Event N: {type: "TEXT_MESSAGE_END"}                         |    |
  |  +-------------------------------------------------------------+    |
  |                          |                                          |
  |                          v                                          |
  |  Step 7: Frontend Updates                                           |
  |  +-------------------------------------------------------------+    |
  |  | - Receives SSE events in real-time                          |    |
  |  | - Updates UI progressively (streaming text)                 |    |
  |  | - Displays complete response                                |    |
  |  | - Ready for next user input                                 |    |
  |  +-------------------------------------------------------------+    |
  |                                                                     |
  +---------------------------------------------------------------------+
```

Quick Example:

```
// Frontend (Next.js)

  return (
    <CopilotKit runtimeUrl="/api/copilotkit">
      <CopilotChat
        instructions="You are a helpful customer support agent."
      />
    </CopilotKit>
  );
}
```

```python
# Backend (Python)
from fastapi import FastAPI
from ag_ui_adk import ADKAgent, add_adk_fastapi_endpoint
from google.adk.agents import Agent

app = FastAPI()

adk_agent = Agent(name="support", model="gemini-2.0-flash-exp")
agent = ADKAgent(
    adk_agent=adk_agent,
    app_name="support_app",
    user_id="user",
    use_in_memory_services=True
)

add_adk_fastapi_endpoint(app, agent, path="/api/copilotkit")
```

Covered in: Tutorial 30 (Next.js), Tutorial 31 (Vite), Tutorial 35 (Advanced)

Approach 2: Native ADK API

When to Use:

- Building custom UI frameworks (Vue, Svelte, Angular)
- Need full control over transport layer
- Want to minimize dependencies
- Building mobile apps (React Native, Flutter)

Architecture:

Quick Example:

```
// Frontend (Any framework)
const response = await fetch("http://localhost:8000/run", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({
    session_id: "user-123",
    user_content: [{ text: "What is ADK?" }],
  }),
});

const result = await response.json();
console.log(result.agent_content);
```

```
# Backend (Python)
from google.adk.agents import Agent

# Create ADK agent
agent = Agent(
    model='gemini-2.0-flash-exp',
    name='my_agent',
    instruction='You are a helpful assistant that provides clear and concise a
)

# For web server deployment, use: adk web agent.py
# Or integrate with FastAPI/Flask for custom HTTP endpoints
```

Covered in: Tutorial 14 (Streaming & SSE), Tutorial 29 (this tutorial)

Approach 3: Direct Python Integration

When to Use:

- Building data apps with Streamlit
- Internal tools and dashboards
- ML/AI workflows
- Python-only stack

Architecture:

```
┌─────────────────────────────────────────────────────────┐
│                                                         │
│  Streamlit App (Python)                                 │
│  ├─ st.chat_message()                                   │
│  ├─ st.chat_input()                                     │
│  └─ Direct ADK integration (in-process)                 │
│                                                         │
│         ↕ (No HTTP - direct Python calls)               │
│                                                         │
│  Your ADK Agent                                         │
│  └─ In-process execution                                │
│                                                         │
└─────────────────────────────────────────────────────────┘
```

Quick Example:

```
import streamlit as st
import asyncio
from google.adk.agents import Agent
from google.adk.runners import InMemoryRunner
from google.genai import types

# Initialize agent
agent = Agent(
    model='gemini-2.0-flash-exp',
    name='data_analyst',
    instruction='You are an expert data analyst who helps users understand the
)

# Initialize runner
runner = InMemoryRunner(agent=agent, app_name='streamlit_app')

async def get_response(prompt: str, session_id: str):
    """Get agent response with proper async pattern."""
    # Create session
    session = await runner.session_service.create_session(
        app_name='streamlit_app',
        user_id='user1'
    )

    # Run query with async iteration
    new_message = types.Content(
        role='user',
        parts=[types.Part(text=prompt)]
    )

    response_text = ""
    async for event in runner.run_async(
        user_id='user1',
        session_id=session.id,
        new_message=new_message
    ):
        if event.content and event.content.parts:
            response_text += event.content.parts[0].text

    return response_text

# Streamlit UI
if prompt := st.chat_input("Ask me about your data"):
    st.chat_message("user").write(prompt)

    # Get response
```

```
    response = asyncio.run(get_response(prompt, 'session1'))
    st.chat_message("assistant").write(response)
```

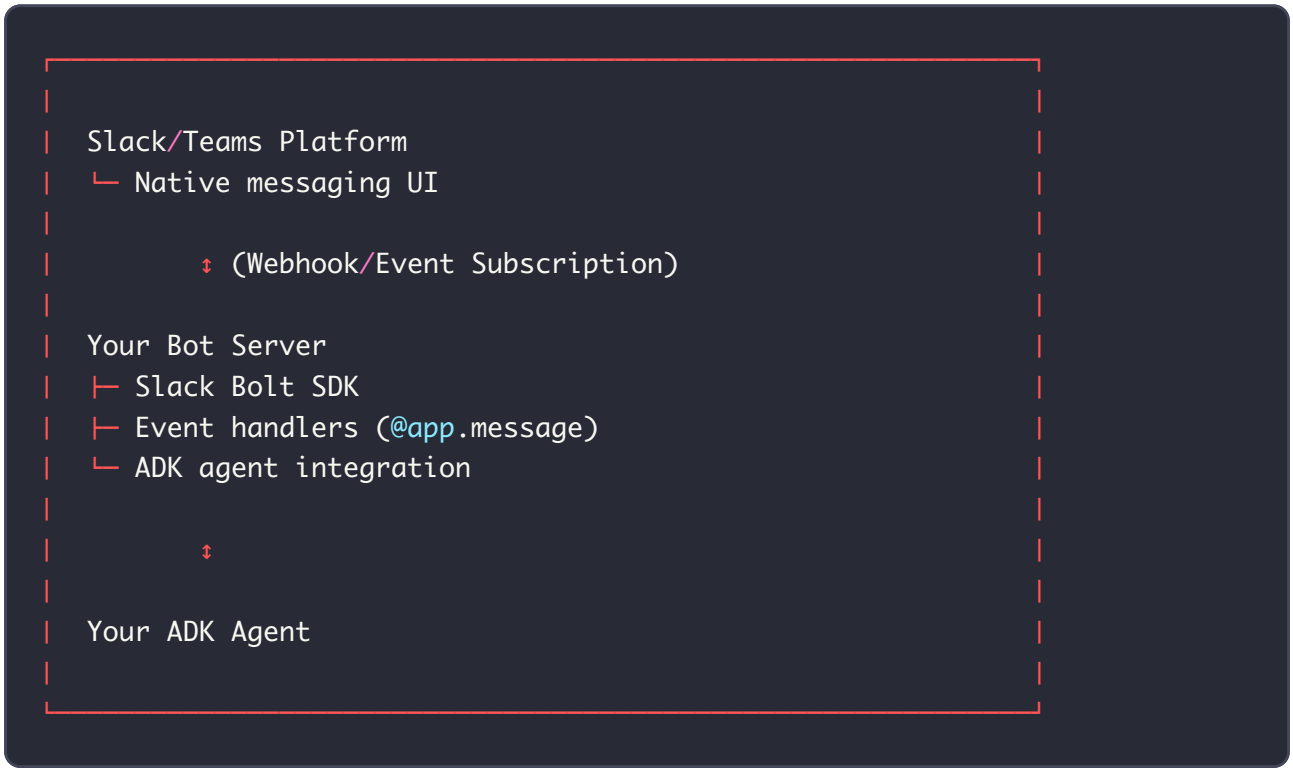
Covered in: Tutorial 32 (Streamlit)

Approach 4: Messaging Platform Integration

When to Use:

- Building team collaboration tools
- Slack/Microsoft Teams bots
- Enterprise internal tools
- Need native platform UX

Architecture:



Quick Example:

```python
from slack_bolt import App
from google.adk.agents import Agent
from google.adk.runners import InMemoryRunner
from google.genai import types
import asyncio

app = App(token="xoxb-...")

# Initialize agent once at startup
agent = Agent(
    model='gemini-2.0-flash-exp',
    name='support_agent',
    instruction='You are a helpful Slack support bot that assists team members
)

# Initialize runner
runner = InMemoryRunner(agent=agent, app_name='slack_bot')

async def get_agent_response(user_id: str, channel_id: str, text: str):
    """Get agent response with proper async pattern."""
    # Create session
    session = await runner.session_service.create_session(
        app_name='slack_bot',
        user_id=user_id
    )

    # Run query with async iteration
    new_message = types.Content(
        role='user',
        parts=[types.Part(text=text)]
    )

    response_text = ""
    async for event in runner.run_async(
        user_id=user_id,
        session_id=session.id,
        new_message=new_message
    ):
        if event.content and event.content.parts:
            response_text += event.content.parts[0].text

    return response_text

@app.message("")
def handle_message(message, say):
    # Get agent response
```

```
    response = asyncio.run(get_agent_response(
        message['user'],
        message['channel'],
        message['text']
    ))

    # Reply in Slack thread
    say(response, thread_ts=message['ts'])

app.start(port=3000)
```

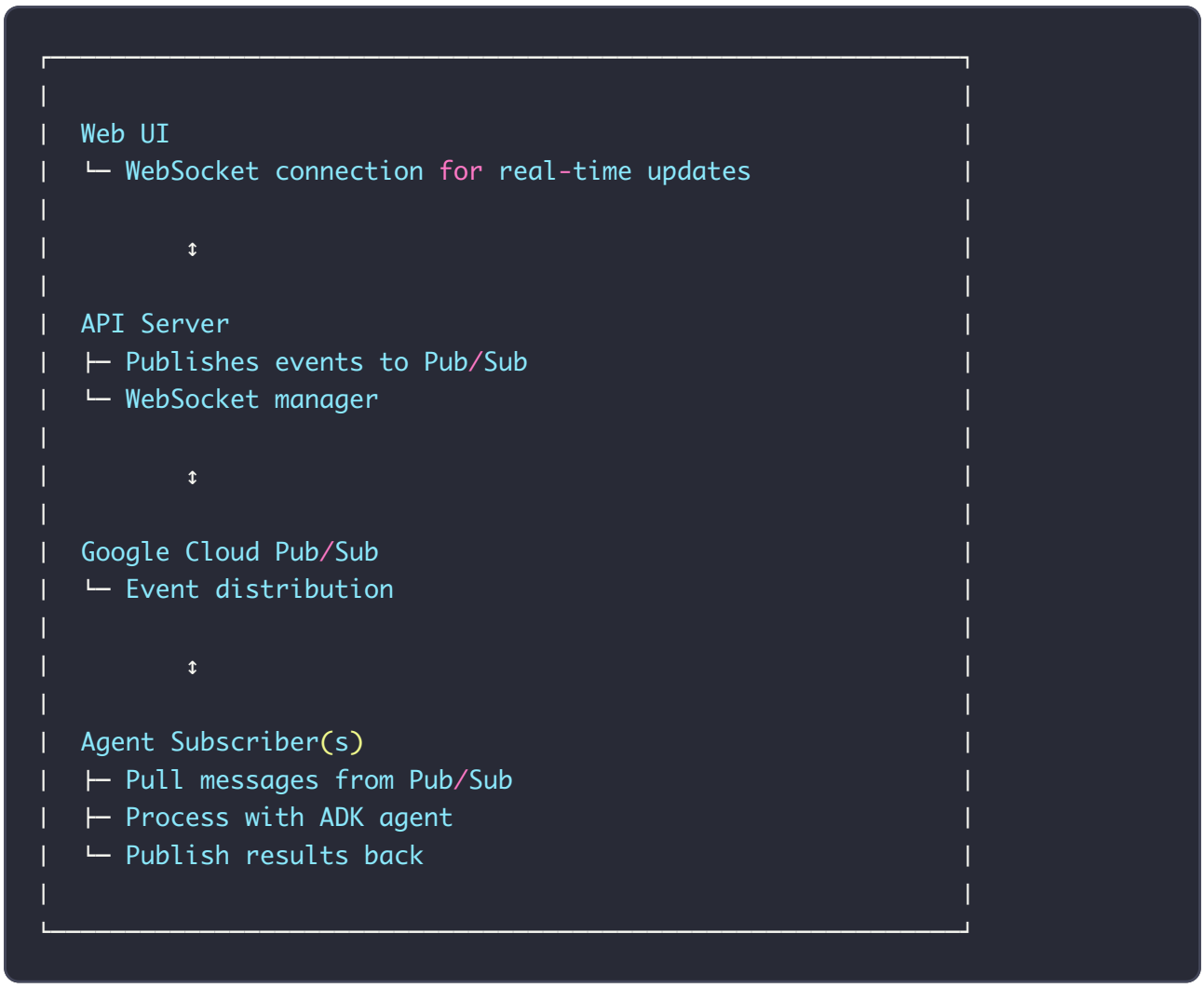
Covered in: Tutorial 33 (Slack)

Approach 5: Event-Driven Architecture

When to Use:

- High-scale systems (millions of events)
- Asynchronous processing
- Multiple subscribers (fan-out)
- Decoupled architectures

Architecture:



Quick Example:

```python
from google.cloud import pubsub_v1
from google import genai

# Publisher
publisher = pubsub_v1.PublisherClient()
topic_path = publisher.topic_path('my-project', 'agent-requests')

# Publish event
publisher.publish(topic_path, data=b'Process document X')

# Initialize agent once at startup (outside callback)
from google.adk.agents import Agent
from google.adk.runners import InMemoryRunner
from google.genai import types
import asyncio

agent = Agent(
    model='gemini-2.0-flash-exp',
    name='doc_processor',
    instruction='You process documents and extract key information.'
)

# Initialize runner
runner = InMemoryRunner(agent=agent, app_name='pubsub_processor')

async def process_message(message_text: str, message_id: str):
    """Process message with proper async pattern."""
    # Create session
    session = await runner.session_service.create_session(
        app_name='pubsub_processor',
        user_id='system'
    )

    # Run query with async iteration
    new_message = types.Content(
        role='user',
        parts=[types.Part(text=message_text)]
    )

    async for event in runner.run_async(
        user_id='system',
        session_id=session.id,
        new_message=new_message
    ):
        if event.content and event.content.parts:
            # Process event (e.g., publish result)
```

```
            print(event.content.parts[0].text)

# Subscriber
subscriber = pubsub_v1.SubscriberClient()
subscription_path = subscriber.subscription_path('my-project', 'agent-sub')

def callback(message):
    # Process message
    asyncio.run(process_message(message.data.decode(), message.message_id))

    # Acknowledge
    message.ack()

subscriber.subscribe(subscription_path, callback=callback)
```

Covered in: Tutorial 34 (Pub/Sub)

Quick Start: Your First AG-UI Integration

Let's build a simple ADK agent with AG-UI in **under 10 minutes**!

Prerequisites

```
# Python 3.9+
python --version

# Node.js 18+
node --version

# Google AI API Key
```

Step 1: Create Backend (Python)

```
# Create project
mkdir adk-quickstart && cd adk-quickstart
mkdir agent && cd agent

# Create virtual environment
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate

# Install dependencies
pip install google-genai fastapi uvicorn ag-ui-adk python-dotenv
```

Create `agent/agent.py`:

```
"""Simple ADK agent with AG-UI integration."""

import os
from dotenv import load_dotenv
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from ag_ui_adk import ADKAgent, add_adk_fastapi_endpoint
from google.adk.agents import Agent
import uvicorn

# Load environment variables
load_dotenv()

# Create ADK agent
adk_agent = Agent(
    name="quickstart_agent",
    model="gemini-2.0-flash-exp",
    instruction="""You are a helpful AI assistant powered by Google ADK.

Your role:
- Answer questions clearly and concisely
- Be friendly and professional
- Provide accurate information
- If you don't know something, say so

Guidelines:
- Keep responses under 3 paragraphs unless more detail is requested
- Use markdown formatting for better readability"""
)

# Wrap with ADKAgent middleware
agent = ADKAgent(
    adk_agent=adk_agent,
    app_name="quickstart_demo",
    user_id="demo_user",
    session_timeout_seconds=3600,
    use_in_memory_services=True
)

# Export for testing
root_agent = adk_agent

# Initialize FastAPI
app = FastAPI(title="ADK Quickstart API")

# Enable CORS for frontend
```

```
app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:5173"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

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

# Health check endpoint
@app.get("/health")
def health_check():
    return {"status": "healthy", "agent": "quickstart_agent"}

if __name__ == "__main__":
    uvicorn.run(app, host="0.0.0.0", port=8000, reload=True)
```

Create `agent/.env.example`:

```
# Google AI API Key (required)
# Get your free key at: https://aistudio.google.com/app/apikey
GOOGLE_API_KEY=your_api_key_here

# Optional configuration
PORT=8000
HOST=0.0.0.0
```

Configure and run backend:

```
# Copy environment template
cp .env.example .env

# Edit .env and add your API key
# Then run the backend
python agent.py
```

Step 2: Create Frontend (React + Vite)

```
# In new terminal, from project root
cd ..
npm create vite@latest frontend -- --template react-ts
cd frontend

# Install dependencies (Tailwind CSS for styling)
npm install
npm install tailwindcss postcss autoprefixer
npx tailwindcss init -p
```

Create `frontend/tailwind.config.js`:

```
/** @type {import('tailwindcss').Config} */

  content: [
    "./index.html",
    "./src/**/*.{js,ts,jsx,tsx}",
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

Update `frontend/src/App.css`:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Update `frontend/src/App.tsx` (simplified custom UI without CopilotKit components):

```
import "./App.css";

interface Message {
  role: "user" | "assistant";
  content: string;
}

function App() {
  const [messages, setMessages] = useState<Message[]>([
    {
      role: "assistant",
      content: "Hi! I'm powered by Google ADK. Ask me anything!",
    },
  ]);
  const [input, setInput] = useState("");
  const [isLoading, setIsLoading] = useState(false);

  const sendMessage = async (e: React.FormEvent) => {
    e.preventDefault();
    if (!input.trim() || isLoading) return;

    const userMessage: Message = { role: "user", content: input };
    setMessages((prev) => [...prev, userMessage]);
    setInput("");
    setIsLoading(true);

    try {
      const response = await fetch("http://localhost:8000/api/copilotkit", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({
          threadId: "quickstart-thread",
          runId: `run-${Date.now()}`,
          messages: [...messages, userMessage].map((m, i) => ({
            id: `msg-${i}`,
            role: m.role,
            content: m.content,
          })),
        }),
      });

      if (!response.ok) throw new Error(`HTTP ${response.status}`);

      // Handle streaming response
      const reader = response.body?.getReader();
      const decoder = new TextDecoder();
```

```
      let fullContent = "";

      if (reader) {
        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: ")) {
              try {
                const jsonData = JSON.parse(line.slice(6));
                if (jsonData.type === "TEXT_MESSAGE_CONTENT") {
                  fullContent += jsonData.delta;
                  setMessages((prev) => {
                    const newMessages = [...prev];
                    const lastMsg = newMessages[newMessages.length - 1];
                    if (lastMsg?.role === "assistant") {
                      lastMsg.content = fullContent;
                    } else {
                      newMessages.push({ role: "assistant", content: fullConte
                    }
                    return newMessages;
                  });
                }
              } catch (e) {
                // Skip invalid JSON
              }
            }
          }
        }
      }
    } catch (error) {
      console.error("Error:", error);
      setMessages((prev) => [
        ...prev,
        { role: "assistant", content: "Error: Could not get response" },
      ]);
    } finally {
      setIsLoading(false);
    }
  };

  return (
    <div className="flex flex-col h-screen bg-gray-50">
```

```
      {/* Header */}
      <header className="bg-white border-b shadow-sm">
        <div className="max-w-4xl mx-auto px-6 py-4">
          <h1 className="text-xl font-bold">ADK Quickstart</h1>
          <p className="text-sm text-gray-600">Gemini 2.0 Flash</p>
        </div>
      </header>

      {/* Chat Messages */}
      <main className="flex-1 overflow-y-auto">
        <div className="max-w-4xl mx-auto px-6 py-8">
          {messages.map((message, index) => (
            <div key={index} className="mb-6">
              <div className={`${message.role === "user" ? "text-blue-600" : "
                <strong>{message.role === "user" ? "You" : "Assistant"}:</stro
                <p>{message.content}</p>
              </div>
            </div>
          ))}
          {isLoading && <div className="text-gray-500">Thinking...</div>}
        </div>
      </main>

      {/* Input Form */}
      <footer className="bg-white border-t shadow-lg">
        <div className="max-w-4xl mx-auto px-6 py-4">
          <form onSubmit={sendMessage} className="flex gap-3">
            <input
              type="text"
              value={input}
              onChange={(e) => setInput(e.target.value)}
              placeholder="Type your message..."
              disabled={isLoading}
              className="flex-1 px-4 py-2 border rounded-lg"
            />
            <button
              type="submit"
              disabled={isLoading || !input.trim()}
              className="px-6 py-2 bg-blue-600 text-white rounded-lg"
            >
              Send
            </button>
          </form>
        </div>
      </footer>
    </div>
```

```
  );
}
```

Run frontend:

```
npm run dev
```

Step 3: Test It

1. Open http://localhost:5173 (http://localhost:5173) in your browser
2. You'll see a chat interface
3. Type: "What is Google ADK?"
4. The agent responds using Gemini!

🎉 **Congratulations! You just built your first ADK UI integration!**

Step 4: Explore the Complete Implementation

The full working implementation with production-ready features is available at:

```
cd tutorial_implementation/tutorial29
```

What's included in the full implementation:

- ✔️ Enhanced backend with middleware for CopilotKit compatibility
- ✔️ Production-ready frontend with Tailwind CSS styling
- ✔️ Comprehensive test suite (15+ tests)
- ✔️ Development workflow with `make` commands
- ✔️ Environment configuration and error handling
- ✔️ Health check and monitoring endpoints

Quick commands:

```
# Setup and run
make setup          # Install all dependencies
make dev            # Start backend + frontend

# Testing
make test           # Run test suite
make demo           # Show example prompts
```

Decision Framework

Choosing the Right Approach

Use this decision tree to select the best integration approach:

```
START
  |
  ├─ Building a web app? ── YES ─┐
  |                              |
  |                              ├─ Using React/Next.js? ── YES ─→ AG-UI P
  |                              |                                 (Tutor
  |                              |
  |                              └─ Using Vue/Svelte/Angular? ─→ Native API
  |                                                              (Tutorial
  |
  ├─ Building a data app? ── YES ─→ Streamlit Direct Integration 📊
  |                                 (Tutorial 32)
  |
  ├─ Building a team bot? ── YES ─→ Slack/Teams Integration 💬
  |                                 (Tutorial 33)
  |
  └─ Need high scale? ── YES ────→ Event-Driven (Pub/Sub) 🚀
                                    (Tutorial 34)
```

Detailed Comparison

AG-UI Protocol vs Native API

Factor	AG-UI Protocol	Native API
Setup Time	⚡ 10 minutes	🔨 1-2 hours
UI Components	✔️ Pre-built (`<CopilotChat>`)	❌ Build yourself
TypeScript Support	✔️ Full type safety	⚠️ Manual types
Framework	React/Next.js only	Any framework
Dependencies	CopilotKit + ag_ui_adk	None (just ADK)
Documentation	✔️ Extensive	✔️ Good
Production Ready	✔️ Yes (271 tests)	✔️ Yes
Customization	🔶 Medium (theme, props)	✔️ Full control

Recommendation: Use **AG-UI Protocol** for React/Next.js apps. Use **Native API** for other frameworks or when you need full control.

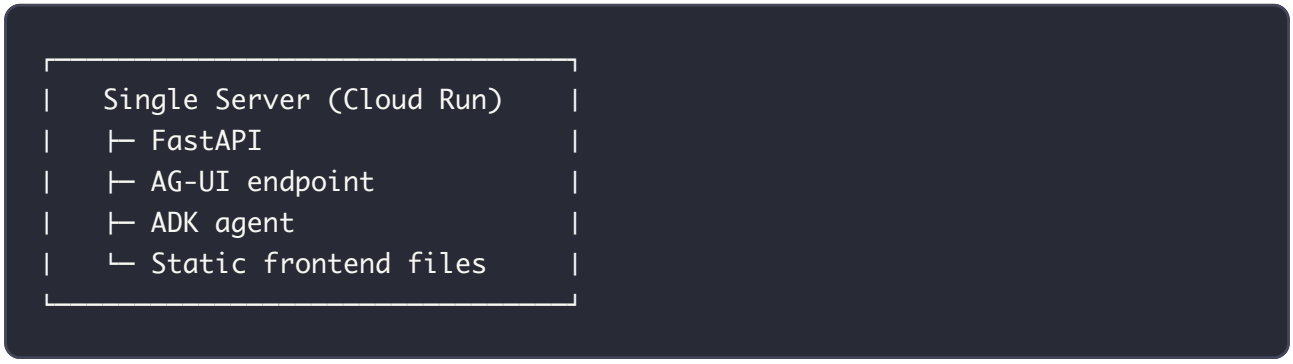
Web vs Python vs Messaging

Use Case	Best Approach	Why?
Customer-facing SaaS	AG-UI (Next.js)	Production-ready, scalable, great UX
Internal data tools	Streamlit	Fast dev, Python-only, built-in UI
Team collaboration	Slack/Teams	Native UX, no custom UI needed
Document processing	Pub/Sub	Async, scalable, decoupled
Mobile app	Native API	Framework-agnostic

Architecture Patterns

Pattern 1: Monolith (Quick Start)

Best for: Prototypes, MVPs, small teams



Pros: Simple deployment, low cost
Cons: Limited scalability

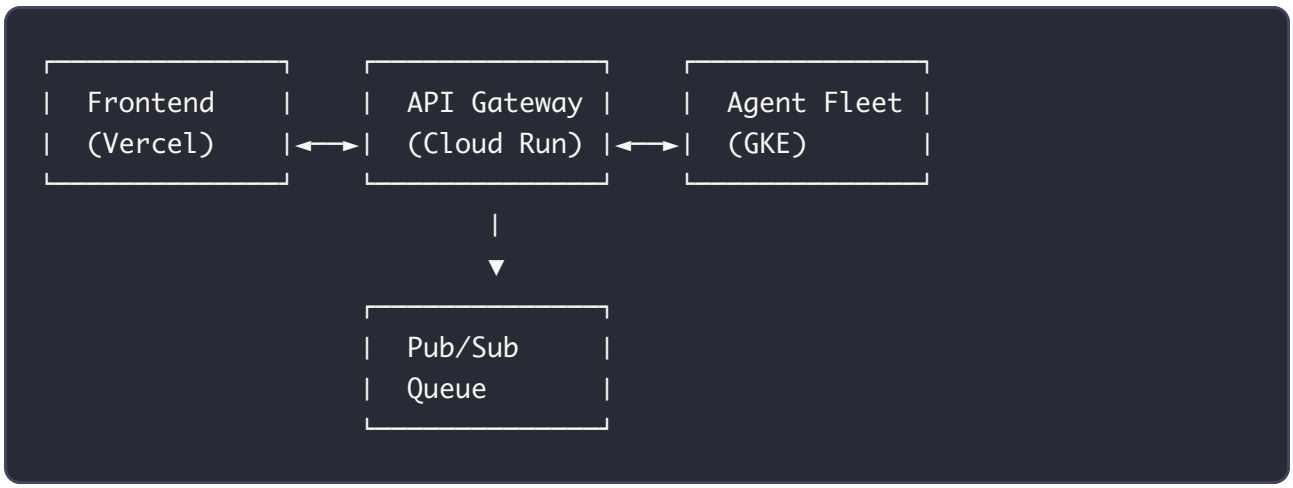
Pattern 2: Separated Frontend/Backend (Recommended)

Best for: Production apps, scaling teams

Pros: Independent scaling, CDN for frontend
Cons: CORS configuration needed

Pattern 3: Microservices (Enterprise)

Best for: Large teams, high scale



Pros: Unlimited scale, fault isolation
Cons: Complex infrastructure

Best Practices

1. Session Management

Always persist agent state for conversation continuity:

Implementation examples:

```python
from google.adk.agents import Agent
from google.adk.runners import InMemoryRunner
from google.genai import types
import asyncio

# ❌ Bad: New agent every request (loses context)
@app.post("/chat")
async def chat_bad(message: str):
    agent = Agent(
        model='gemini-2.0-flash-exp',
        name='support_agent',
        instruction='You are a helpful support agent'
    )
    runner = InMemoryRunner(agent=agent, app_name='support')
    session = await runner.session_service.create_session(
        app_name='support', user_id='user1'
    )

    new_message = types.Content(role='user', parts=[types.Part(text=message)])
    response_text = ""
    async for event in runner.run_async(
        user_id='user1',
        session_id=session.id,
        new_message=new_message
    ):
        if event.content and event.content.parts:
            response_text += event.content.parts[0].text

    return response_text

# ✔️ Good: Initialize agent and runner once, reuse for conversations
agent = Agent(
    model='gemini-2.0-flash-exp',
    name='support_agent',
    instruction='You are a helpful support agent with conversation memory'
)
runner = InMemoryRunner(agent=agent, app_name='support')

@app.post("/chat")
async def chat(user_id: str, session_id: str, message: str):
    # Create or get session
    session = await runner.session_service.create_session(
        app_name='support',
        user_id=user_id
    )
```

```
    # Runner manages conversation history with session_id
    new_message = types.Content(role='user', parts=[types.Part(text=message)])
    response_text = ""
    async for event in runner.run_async(
        user_id=user_id,
        session_id=session.id,
        new_message=new_message
    ):
        if event.content and event.content.parts:
            response_text += event.content.parts[0].text

    return response_text
```

2. Error Handling

Gracefully handle agent failures:

```
from fastapi import HTTPException

@app.post("/chat")
async def chat(message: str):
    try:
        response = await agent.send_message(message)
        return {"response": response.text}
    except Exception as e:
        # Log error for debugging
        logger.error(f"Agent error: {e}")

        # Return friendly error to user
        raise HTTPException(
            status_code=500,
            detail="I'm having trouble processing that request. Please try aga
        )
```

3. Rate Limiting

Protect your API from abuse:

```python
from slowapi import Limiter
from slowapi.util import get_remote_address

limiter = Limiter(key_func=get_remote_address)
app.state.limiter = limiter

@app.post("/chat")
@limiter.limit("10/minute")  # 10 requests per minute
async def chat(request: Request, message: str):
    # ... agent logic
    pass
```

4. Streaming for Better UX

Stream responses for long-running agents:

```
+-------------------------------------------------------------------+
|                 STREAMING VS NON-STREAMING                        |
+-------------------------------------------------------------------+
|                                                                   |
|  Non-Streaming (Traditional)                                      |
|  +-------------------------------------------------------------+  |
|  | User: "Explain quantum computing"                           |  |
|  |                                                             |  |
|  | [Wait... Wait... Wait... 10 seconds]                        |  |
|  |                                                             |  |
|  | Agent: [Complete response appears all at once]              |  |
|  |   "Quantum computing is a revolutionary..."                 |  |
|  +-------------------------------------------------------------+  |
|                                                                   |
|  Streaming (Better UX)                                            |
|  +-------------------------------------------------------------+  |
|  | User: "Explain quantum computing"                           |  |
|  |                                                             |  |
|  | Agent: "Quantum..."                    [Instant feedback]   |  |
|  | Agent: "Quantum computing is..."       [Progressive]        |  |
|  | Agent: "Quantum computing is a..."     [User stays]         |  |
|  | Agent: "Quantum computing is a revo..."[engaged]            |  |
|  | Agent: [Complete] "...revolutionary technology"             |  |
|  +-------------------------------------------------------------+  |
|                                                                   |
|  Benefits:                                                        |
|  - Immediate feedback (reduces perceived latency)                 |
|  - Users stay engaged (see progress)                              |
|  - Can cancel early if not relevant                               |
|  - Better mobile experience                                       |
|                                                                   |
+-------------------------------------------------------------------+
```

Implementation examples:

```
// Frontend: Stream responses
const { messages, sendMessage, isLoading } = useCopilotChat({
  stream: true, // Enable streaming
});

// User sees partial responses as agent thinks
```

```
# Backend: Enable streaming
agent = ADKAgent(
    name="streaming_agent",
    model="gemini-2.0-flash-exp",
    stream=True  # Return partial responses
)
```

5. Monitoring & Observability

Track agent performance:

```
from opentelemetry import trace
from opentelemetry.exporter.cloud_trace import CloudTraceSpanExporter

# Set up tracing
tracer = trace.get_tracer(__name__)

@app.post("/chat")
async def chat(message: str):
    with tracer.start_as_current_span("agent_chat"):
        span = trace.get_current_span()
        span.set_attribute("message_length", len(message))

        response = await agent.send_message(message)

        span.set_attribute("response_length", len(response.text))
        return response
```

Next Steps

Where to Go From Here

Now that you understand the UI integration landscape, choose your path:

For Web Developers

→ **Tutorial 30**: Next.js 15 + ADK Integration (AG-UI)
Build a production-ready customer support chatbot with Next.js 15 and deploy to Vercel.

→ **Tutorial 31**: React Vite + ADK Integration (AG-UI)
Create a lightweight data analysis dashboard with React Vite.

→ **Tutorial 35**: AG-UI Deep Dive - Building Custom Components
Master advanced AG-UI features: generative UI, human-in-the-loop, custom components.

For Python/Data Engineers

→ **Tutorial 32**: Streamlit + ADK Integration
Build interactive data apps with direct Python integration.

For DevOps/Enterprise Teams

→ **Tutorial 33**: Slack Bot Integration with ADK
Create team collaboration bots for Slack.

→ **Tutorial 34**: Google Cloud Pub/Sub + Event-Driven Agents
Design scalable, event-driven agent architectures.

Additional Resources

Official Documentation:

- Google ADK Documentation (https://google.github.io/adk-docs/)
- AG-UI Protocol Docs (https://docs.copilotkit.ai)
- CopilotKit GitHub (https://github.com/CopilotKit/CopilotKit)

Sample Code:

- ADK Samples Repository (https://github.com/google/adk-samples)
- gemini-fullstack Example (https://github.com/google/adk-samples/tree/main/gemini-fullstack)

Community:

- CopilotKit Discord (https://discord.gg/copilotkit)
- Google AI Community (https://discuss.ai.google.dev)

Summary

Key Takeaways

✔️ **Multiple Integration Options**: AG-UI Protocol, Native API, Direct Python, Messaging, Pub/Sub
✔️ **AG-UI Protocol**: Official, production-ready solution for React/Next.js
✔️ **Decision Framework**: Choose based on framework, scale, and use case
✔️ **Quick Start**: Get running in under 10 minutes
✔️ **Best Practices**: Session management, error handling, streaming, monitoring

What's Next

You now have a comprehensive understanding of ADK UI integration. The next tutorials will dive deep into each integration approach with production-ready examples.

Ready to build? Start with Tutorial 30 for web apps or Tutorial 32 for data apps!

🎉 **Tutorial 29 Complete!**

Next: Tutorial 30: Next.js 15 + ADK Integration (./30_nextjs_adk_integration.md)

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

Generated on 2025-10-19 17:57:14 from 29_ui_integration_intro.md

Source: Google ADK Training Hub