Tutorial 32: Streamlit ADK Integration - Python Data Apps

Difficulty: intermediate
Reading Time: 1.5 hours
Tags: ui, streamlit, python, data-science, dashboard
Description: Build data science applications with Streamlit and ADK agents for interactive dashboards, analysis tools, and data-driven interfaces.

Tutorial 32: Streamlit + ADK - Build Data Analysis Apps in Pure Python

Time: 45 minutes | **Level**: Intermediate | **Language**: Python only

---

Why This Matters

Building data apps shouldn't require learning JavaScript, React, or managing separate frontend/backend services. **Streamlit + ADK** lets you build production-grade data analysis apps in pure Python.

The Problem You're Solving

```
Without this approach:
├─ Learn React/Vue/Angular
├─ Set up TypeScript
├─ Manage separate backend API
├─ Deploy two services
├─ Handle CORS, authentication, etc.
└─ Takes weeks to get right 😫

With Streamlit + ADK:
├─ Pure Python only
├─ In-process AI agent (no HTTP)
├─ One file = complete app
├─ Deploy in 2 minutes
└─ Works immediately 🚀
```

What You'll Build

A **data analysis chatbot** that:

- Accepts CSV file uploads
- Chats with your data naturally
- Generates charts with matplotlib/plotly
- Deploys to the cloud with one command
- Runs completely in Python

Visual preview:

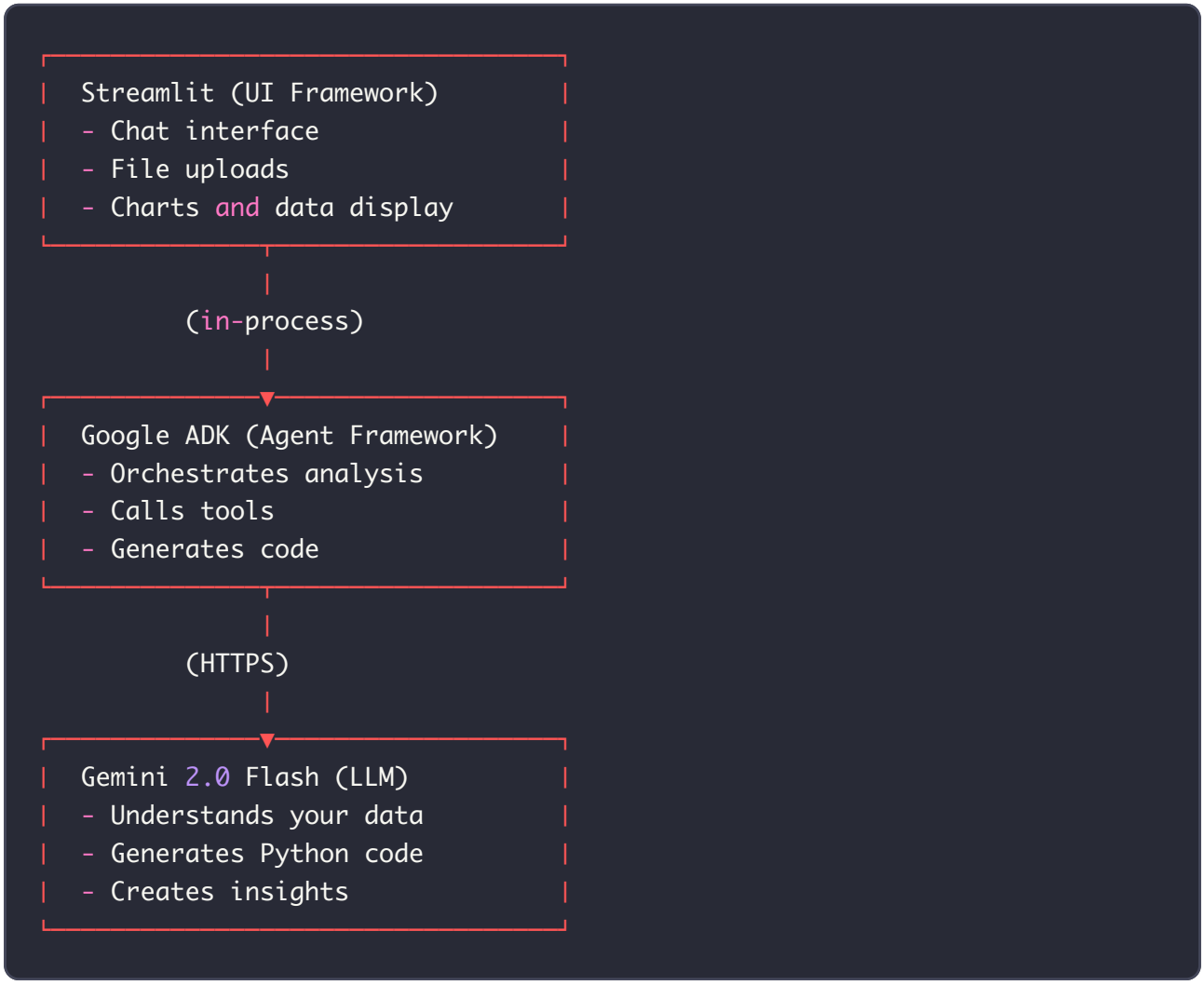
```
User: "What are my top 5 customers?"
      ↓
[🔍 Processing... analyzing data...]
      ↓
Bot: "Based on your data:

      Top 5 Customers by Revenue:
      1. Acme Corp - $125,000
      2. Tech Inc - $98,500
      ..."
```

How It Works

The Tech Stack

Three simple pieces:



Why This Approach?

Need	Solution	Benefit
UI	Streamlit	No HTML/CSS, pure Python
AI Logic	ADK	No HTTP overhead
LLM	Gemini	Blazing fast, smart
Deployment	One service	Simple, reliable

Getting Started (5 Minutes)

Prerequisites

```
# Check Python version
python --version  # Should be 3.9 or higher
```

Need a Google API key?

1. Visit Google AI Studio (https://makersuite.google.com/app/apikey)
2. Click "Get API key"
3. Copy it (keep it safe!)

Run the Demo

```
cd tutorial_implementation/tutorial32

# Setup once
make setup

# Create config
cp .env.example .env
# Edit .env and paste your API key

# Start
make dev
```

Open http://localhost:8501 (http://localhost:8501) and you're done! 🚀

Building Your App

The Minimal Example

Here's the bare minimum to get started (`app.py`):

```python
import os
import streamlit as st
import pandas as pd
from google import genai

# Setup
st.set_page_config(page_title="Data Analyzer", page_icon="📊", layout="wide")
client = genai.Client(api_key=os.getenv("GOOGLE_API_KEY"))

# State
if "messages" not in st.session_state:
    st.session_state.messages = []
if "df" not in st.session_state:
    st.session_state.df = None

# UI
st.title("📊 Data Analyzer")

# Upload
with st.sidebar:
    file = st.file_uploader("CSV file", type=["csv"])
    if file:
        st.session_state.df = pd.read_csv(file)

# Display messages
for msg in st.session_state.messages:
    with st.chat_message(msg["role"]):
        st.markdown(msg["content"])

# Chat
if prompt := st.chat_input("Ask about your data..."):
    st.session_state.messages.append({"role": "user", "content": prompt})

    with st.chat_message("user"):
        st.markdown(prompt)

    # Get response
    with st.chat_message("assistant"):
        with st.status("Analyzing...", expanded=False) as status:
            status.write("Reading data...")

            # Add data context
            context = f"Dataset: {st.session_state.df.shape[0]} rows, "
            context += f"{st.session_state.df.shape[1]} columns"

            status.write("Thinking...")
```

```
            response = client.models.generate_content_stream(
                model="gemini-2.0-flash",
                contents=[{"role": "user", "parts": [{"text": context}]}],
            )

            full_text = ""
            for chunk in response:
                if chunk.text:
                    full_text += chunk.text

            status.update(label="Done!", state="complete", expanded=False)

        st.markdown(full_text)
        st.session_state.messages.append({"role": "assistant", "content": full
```

That's it! Run `streamlit run app.py` and you have a working data analyzer. 🎉

Key Concepts

1. Streamlit Caching

Avoid recomputing expensive operations:

```
@st.cache_resource  # Computed once, reused forever
def get_client():
    return genai.Client(api_key=os.getenv("GOOGLE_API_KEY"))

@st.cache_data  # Recompute on data change
def load_csv(uploaded_file):
    return pd.read_csv(uploaded_file)
```

2. Session State

Store data that persists across reruns:

```
# Initialize on first run
if "messages" not in st.session_state:
    st.session_state.messages = []

# Use throughout app
st.session_state.messages.append({"role": "user", "content": prompt})
```

3. Status Container

Show progress to users (Streamlit best practice):

```
with st.status("Processing...", expanded=False) as status:
    status.write("Step 1: Loading data")
    # ... do work ...

    status.write("Step 2: Analyzing")
    # ... more work ...

    status.update(label="Complete!", state="complete")
```

Understanding the Architecture

Component Diagram

Key Differences from Next.js/Vite:

Aspect	Streamlit	Next.js/Vite
Architecture	Single Python process	Frontend + Backend
Communication	In-process function calls	HTTP/WebSocket
Latency	~0ms (in-process)	~50-100ms (network)
Deployment	Single service	Two services
Complexity	Simple (1 file)	Medium (multiple files)
Use Case	Data tools, internal apps	Production web apps

Request Flow

1. User uploads CSV file

```
# Streamlit handles file upload
uploaded_file = st.file_uploader("Upload CSV")

# Load into pandas
df = pd.read_csv(uploaded_file)

# Store in session state (persists across reruns)
st.session_state.dataframe = df
```

2. User sends message "What are the top 5 customers by revenue?"

3. Streamlit app

```
# Build context with dataset info
context = f"""
Dataset available:
- Columns: {df.columns.tolist()}
- First rows: {df.head(3)}
"""

# Call Gemini directly (in-process!)
response = client.models.generate_content_stream(
    model="gemini-2.0-flash-exp",
    contents=[...],
    config=GenerateContentConfig(
        system_instruction=f"You are a data analyst. {context}"
    )
)
```

4. Gemini API

```
System: You are a data analyst. Dataset has columns: customer, revenue...
User: What are the top 5 customers by revenue?
Model: Based on your data, the top 5 customers are:
1. Acme Corp - $125,000
2. Tech Inc - $98,500
...
```

5. Response streams back

```
# Stream chunks as they arrive
for chunk in response:
    full_response += chunk.text
    message_placeholder.markdown(full_response + "▌")
```

6. User sees response typing in real-time! ⚡

Understanding ADK (Agent Development Kit)

This is where Streamlit + ADK shines. You might be wondering: **"Why use ADK instead of calling Gemini directly?"**

Great question! Let's explore the architecture.

Direct API vs ADK Architecture

Direct Gemini API (Simpler but Limited)

```
# What we showed in the Request Flow above
client = genai.Client(api_key=...)
response = client.models.generate_content(
    model="gemini-2.0-flash",
    contents=[...]
)
```

Pros:

- ✔️ Simple, direct, minimal setup
- ✔️ Works great for basic chat
- ✔️ Full control over prompts

Cons:

- ❌ No tool/function calling orchestration
- ❌ No code execution capabilities
- ❌ Manual prompt engineering
- ❌ No reusable agent patterns

ADK Architecture (Powerful but More Features)

```python
# With ADK Agents
from google.adk.agents import Agent
from google.adk.runners import Runner

# Define your agent with tools
agent = Agent(
    name="data_analysis_agent",
    model="gemini-2.0-flash",
    tools=[analyze_column, calculate_correlation, filter_data]
)

# Create a runner to orchestrate it
runner = Runner(agent=agent, app_name="my_app")

# Execute in Streamlit
async for event in runner.run_async(
    user_id="streamlit_user",
    session_id=session_id,
    new_message=message
):
    # Handle agent responses
    process_event(event)
```

Pros:

- ✔️ Automatic tool/function orchestration
- ✔️ Code execution for dynamic visualizations
- ✔️ Multi-agent coordination
- ✔️ State management across sessions
- ✔️ Error handling and retries
- ✔️ Reusable agent components

Cons:

- ❌ Slightly more setup
- ❌ Need to structure agents properly

When to Use Each

Use Case	Approach	Reason
Simple chat about data	Direct API	Fast, minimal setup
Need tool calling	ADK	Automatic orchestration
Data analysis with tools	ADK	Better structure
Dynamic code execution	ADK	BuiltInCodeExecutor support
Multi-agent workflows	ADK	Multi-agent routing
Production apps	ADK	Better error handling

🎯 **This tutorial uses both**: Level 1-2 show direct API for learning, Level 3+ show ADK for production patterns.

ADK Core Concepts

1. Agents

An **Agent** is an AI entity that can:

- Understand user requests
- Call tools (functions you provide)
- Reason about results
- Generate code and execute it

```
from google.adk.agents import Agent

agent = Agent(
    name="analyzer",
    model="gemini-2.0-flash",
    description="Analyzes data",
    instruction="You are a data analyst. Help users understand their datasets.
    tools=[tool1, tool2, tool3]  # List of functions the agent can call
)
```

2. Tools

Tools are Python functions that agents can call:

```
def analyze_column(column_name: str, analysis_type: str) -> dict:
    """Analyze a specific column."""
    # Your logic here
    return {
        "status": "success",
        "report": "analysis results",
        "data": {...}
    }

# Register with agent
agent = Agent(
    tools=[analyze_column, calculate_correlation, filter_data]
)
```

3. Runners

A **Runner** orchestrates agent execution in Streamlit:

```
from google.adk.runners import Runner
from google.adk.sessions import InMemorySessionService

session_service = InMemorySessionService()
runner = Runner(
    agent=agent,
    app_name="data_analysis_assistant",
    session_service=session_service
)

# Execute in Streamlit
async for event in runner.run_async(
    user_id="streamlit_user",
    session_id=session_id,
    new_message=message
):
    handle_event(event)
```

4. Code Execution

ADK supports **BuiltInCodeExecutor** for dynamic visualization:

```
from google.adk.code_executors import BuiltInCodeExecutor

code_executor = BuiltInCodeExecutor()

viz_agent = Agent(
    name="visualization_agent",
    model="gemini-2.0-flash",
    instruction="Generate Python code for visualizations",
    code_executor=code_executor  # Enable code execution!
)
```

This lets agents:

- Generate Python code
- Execute it safely in a sandbox
- Return visualizations as inline images
- Handle errors gracefully

ADK Architecture Diagram

What ADK Gives You

With ADK, you get:

1. **Automatic Tool Calling**: Agent figures out which tools to use
2. **Streaming Responses**: Events stream back as agent thinks
3. **Code Execution**: Agents can write and run Python
4. **Multi-Agent**: Coordinate multiple specialized agents
5. **State Management**: Session persistence without extra code
6. **Error Handling**: Automatic retries and fallbacks
7. **Type Safety**: Tool parameters with validation

Building Your App - Progressive Examples

Now that you understand the basics and ADK architecture, let's build up complexity step-by-step.

Level 1: Basic Chat (Starting Point) ✓

You already have this - a 50-line app that chats about your data.

Level 2: Add Error Handling & Better Context

Let's improve the minimal example with better error handling and dataset context:

```
import os
import streamlit as st
import pandas as pd
from google import genai

st.set_page_config(page_title="Data Analyzer", page_icon="📊", layout="wide")
client = genai.Client(api_key=os.getenv("GOOGLE_API_KEY"))

# State initialization
if "messages" not in st.session_state:
    st.session_state.messages = []
if "df" not in st.session_state:
    st.session_state.df = None

# Sidebar: File upload
with st.sidebar:
    uploaded_file = st.file_uploader("Upload CSV", type=["csv"])
    if uploaded_file is not None:
        try:
            st.session_state.df = pd.read_csv(uploaded_file)
            st.success(f"✓ Loaded {len(st.session_state.df)} rows")
        except Exception as e:
            st.error(f"Error loading file: {e}")

# Main chat interface
st.title("📊 Data Analyzer")

# Display conversation
for msg in st.session_state.messages:
    with st.chat_message(msg["role"]):
        st.markdown(msg["content"])

# Chat input
if prompt := st.chat_input("Ask about your data..."):
    st.session_state.messages.append({"role": "user", "content": prompt})

    with st.chat_message("user"):
        st.markdown(prompt)

    if st.session_state.df is None:
        with st.chat_message("assistant"):
            response = "Please upload a CSV file first!"
            st.markdown(response)
            st.session_state.messages.append({"role": "assistant", "content":
    else:
        # Build rich context
```

```
        df = st.session_state.df
        context = f"""
Dataset Summary:
- {len(df)} rows × {len(df.columns)} columns
- Columns: {', '.join(df.columns.tolist())}
- Memory: {df.memory_usage(deep=True).sum() / 1024**2:.2f} MB

Data Preview:
{df.head(3).to_string()}
"""

        with st.chat_message("assistant"):
            try:
                with st.status("Analyzing...", expanded=False) as status:
                    status.write("Reading context...")

                    response = client.models.generate_content_stream(
                        model="gemini-2.0-flash",
                        contents=[{"role": "user", "parts": [{"text": f"{conte
                    )

                    full_text = ""
                    for chunk in response:
                        if chunk.text:
                            full_text += chunk.text

                    status.update(label="Complete!", state="complete")

                st.markdown(full_text)
                st.session_state.messages.append({"role": "assistant", "conten

            except Exception as e:
                st.error(f"Error: {e}")
```

What's improved:

- ✓ Better context preparation
- ✓ File upload in sidebar
- ✓ Error handling for missing data
- ✓ Status container for progress
- ✓ Memory usage info

Level 3: Using ADK with Runners

Now let's use actual ADK agents with tools and runners. This is the production-pattern version:

Step 1: Create your agents (`data_analysis_agent/agent.py`):

```
"""
Data Analysis Agent - Main analysis orchestrator
Uses ADK Agent framework with tool calling
"""

from typing import Any, Dict
from google.adk.agents import Agent

def analyze_column(column_name: str, analysis_type: str) -> Dict[str, Any]:
    """Analyze a specific column (summary, distribution, outliers)."""
    try:
        if not column_name:
            return {"status": "error", "report": "Column name required"}

        return {
            "status": "success",
            "report": f"Analysis configured for {column_name}",
            "analysis_type": analysis_type,
            "column_name": column_name,
            "note": "Streamlit app will execute with real data"
        }
    except Exception as e:
        return {"status": "error", "report": str(e)}

def calculate_correlation(
    column1: str, column2: str
) -> Dict[str, Any]:
    """Calculate correlation between columns."""
    try:
        if not column1 or not column2:
            return {"status": "error", "report": "Two columns required"}

        return {
            "status": "success",
            "report": f"Correlation calculation configured",
            "column1": column1,
            "column2": column2
        }
    except Exception as e:
        return {"status": "error", "report": str(e)}

def filter_data(
    column_name: str, operator: str, value: str
) -> Dict[str, Any]:
    """Filter dataset by condition."""
    try:
```

```
        return {
            "status": "success",
            "report": f"Filter: {column_name} {operator} {value}",
            "column_name": column_name,
            "operator": operator,
            "value": value
        }
    except Exception as e:
        return {"status": "error", "report": str(e)}

# Create the ADK Agent
root_agent = Agent(
    name="data_analysis_agent",
    model="gemini-2.0-flash",
    description="Analyzes datasets with tools and insights",
    instruction="""You are an expert data analyst. Your role:
1. Help users understand their datasets
2. Analyze columns and distributions
3. Find correlations and patterns
4. Identify outliers and anomalies
5. Provide actionable insights

Use the available tools to analyze data when needed.
Always explain results clearly and suggest follow-up analyses.""",
    tools=[analyze_column, calculate_correlation, filter_data]
)
```

Step 2: Use the agent in Streamlit (`app.py`):

```
"""
Data Analysis Assistant with ADK Agents
Multi-mode: ADK agents for analysis, Streamlit for UI
"""

import asyncio
import os
import streamlit as st
import pandas as pd
from dotenv import load_dotenv
from google import genai
from google.genai.types import Content, Part
from google.adk.runners import Runner
from google.adk.sessions import InMemorySessionService

# Import your agent
from data_analysis_agent import root_agent

load_dotenv()
st.set_page_config(
    page_title="Data Analysis",
    page_icon="📊",
    layout="wide"
)

# ===== AGENT SETUP =====

@st.cache_resource
def get_runner():
    """Initialize ADK runner for agent execution."""
    session_service = InMemorySessionService()
    return Runner(
        agent=root_agent,
        app_name="data_analysis_assistant",
        session_service=session_service,
    ), session_service

runner, session_service = get_runner()

# ===== INITIALIZE ADK SESSION =====

if "adk_session_id" not in st.session_state:
    async def init_session():
        session = await session_service.create_session(
            app_name="data_analysis_assistant",
            user_id="streamlit_user"
```

```
        )
        return session.id

    st.session_state.adk_session_id = asyncio.run(init_session())

# ====== STATE ======

if "messages" not in st.session_state:
    st.session_state.messages = []
if "df" not in st.session_state:
    st.session_state.df = None

# ====== UI =====

st.title("📊 Data Analysis Assistant (ADK)")

# Sidebar
with st.sidebar:
    st.header("📁 Upload Data")
    uploaded_file = st.file_uploader(
        "Choose a CSV file",
        type=["csv"]
    )

    if uploaded_file is not None:
        try:
            st.session_state.df = pd.read_csv(uploaded_file)
            st.success(f"✔️ {len(st.session_state.df)} rows loaded")

            with st.expander("📋 Preview"):
                st.dataframe(
                    st.session_state.df.head(5),
                    use_container_width=True
                )
        except Exception as e:
            st.error(f"Error: {e}")

# Chat display
for message in st.session_state.messages:
    with st.chat_message(message["role"]):
        st.markdown(message["content"])

# Chat input
if prompt := st.chat_input(
    "Ask about your data..." if st.session_state.df is not None
    else "Upload a CSV first",
    disabled=st.session_state.df is None
```

```
):
    st.session_state.messages.append({"role": "user", "content": prompt})

    with st.chat_message("user"):
        st.markdown(prompt)

    # Prepare context
    if st.session_state.df is not None:
        df = st.session_state.df
        context = f"""
**Dataset**: {len(df)} rows × {len(df.columns)} columns
**Columns**: {', '.join(df.columns.tolist())}

**Preview**:
{df.head(3).to_string()}

**User Question**: {prompt}
"""
    else:
        context = f"User: {prompt}"

    with st.chat_message("assistant"):
        response_text = ""

        try:
            with st.status(
                "🔍 Analyzing...",
                expanded=False
            ) as status:
                # Create ADK message
                message = Content(
                    role="user",
                    parts=[Part.from_text(text=context)]
                )

                # Execute agent
                async def run_agent():
                    response = ""
                    async for event in runner.run_async(
                        user_id="streamlit_user",
                        session_id=st.session_state.adk_session_id,
                        new_message=message
                    ):
                        if (event.content and
                            event.content.parts):
                            for part in event.content.parts:
                                if part.text:
```

```
                                response += part.text
                return response

            response_text = asyncio.run(run_agent())
            status.update(
                label="✔️ Done",
                state="complete",
                expanded=False
            )

    except Exception as e:
        response_text = f"❌ Error: {str(e)}"
        st.error(response_text)

    st.markdown(response_text)
    st.session_state.messages.append({
        "role": "assistant",
        "content": response_text
    })
```

Key differences from Level 2:

- ✓ Uses ADK Agent instead of direct API
- ✓ ADK runner orchestrates tool calling
- ✓ Tools automatically called by agent
- ✓ Proper async/await patterns
- ✓ Production-ready error handling
- ✓ Structured Content and Part objects

Advanced: Multi-Agent Systems with ADK

The real power of ADK comes from **multi-agent coordination**. Let's build a system with specialized agents:

Architecture: Analysis Agent + Visualization Agent

```
User Input
    ↓
    ├→ [Analysis Agent]
    |   ├─ analyze_column()
    |   ├─ calculate_correlation()
    |   └─ filter_data()
    |   → Returns insights
    |
    └→ [Visualization Agent]
       ├─ BuiltInCodeExecutor
       ├─ generates Python code
       ├─ executes it safely
       └─ returns charts

Response combines both:
- Insights from Analysis Agent
- Visualizations from Visualization Agent
```

Step 1: Create Visualization Agent

File: `data_analysis_agent/visualization_agent.py`

```
"""
Visualization Agent - Generates dynamic charts with code execution
Uses ADK's BuiltInCodeExecutor to run Python safely
"""

from google.adk.agents import Agent
from google.adk.code_executors import BuiltInCodeExecutor

code_executor = BuiltInCodeExecutor()

visualization_agent = Agent(
    name="visualization_agent",
    model="gemini-2.0-flash",
    description="Generates data visualizations",
    instruction="""You are an expert data visualization specialist.
Your role: Create clear, informative visualizations that help users
understand their data.

**Critical**: You MUST generate Python code that:
1. Loads the DataFrame from provided CSV data
2. Creates visualizations using matplotlib/plotly
3. Saves or returns the chart

**Data Loading Pattern**:
```python
import pandas as pd
from io import StringIO
csv_data = '''[CSV data from context]'''
df = pd.read_csv(StringIO(csv_data))
```

Visualization Examples:

```
import matplotlib.pyplot as plt
plt.figure(figsize=(12, 6))
plt.hist(df['column_name'], bins=30)
plt.title('Distribution of column_name')
plt.show()
```

When asked for visualizations:

1. Don't ask clarifying questions
2. Load the DataFrame from CSV
3. Generate Python code immediately
```

4. Choose appropriate chart types
5. Return publication-ready visualizations""",
   code_executor=code_executor, # Enable code execution!
)

Step 2: Update Main Agent File

File: `data_analysis_agent/agent.py`

```python
"""
Root Agent - Routes between analysis and visualization agents
"""

from typing import Any, Dict
from google.adk.agents import Agent
from google.adk.tools.agent_tool import AgentTool

# Import specialized agents
from .visualization_agent import visualization_agent

def analyze_column(column_name: str, analysis_type: str) -> Dict[str, Any]:
    """Analyze a column."""
    return {
        "status": "success",
        "report": f"Analysis of {column_name}: {analysis_type}",
        "column_name": column_name,
        "analysis_type": analysis_type
    }

def calculate_correlation(column1: str, column2: str) -> Dict[str, Any]:
    """Calculate correlation."""
    return {
        "status": "success",
        "report": f"Correlation between {column1} and {column2}",
        "column1": column1,
        "column2": column2
    }

def filter_data(column: str, operator: str, value: str) -> Dict[str, Any]:
    """Filter dataset."""
    return {
        "status": "success",
        "report": f"Filter: {column} {operator} {value}",
        "column": column,
        "operator": operator,
        "value": value
    }

# Root analysis agent
root_agent = Agent(
```

```
    name="data_analysis_agent",
    model="gemini-2.0-flash",
    description="Data analysis with tools",
    instruction="""You are a data analyst. Help users:
1. Understand their data
2. Find patterns and correlations
3. Identify issues and anomalies
4. Get actionable insights

Use tools to analyze data when appropriate.""",
    tools=[analyze_column, calculate_correlation, filter_data]
)
```

Step 3: Update Streamlit to Support Visualizations

File: `app.py` (modify the agent execution section)

```
# In your chat input handler, after agent execution:

async def run_analysis():
    """Run analysis agent and get response."""
    message = Content(
        role="user",
        parts=[Part.from_text(text=context)]
    )

    response = ""
    async for event in runner.run_async(
        user_id="streamlit_user",
        session_id=st.session_state.adk_session_id,
        new_message=message
    ):
        if event.content and event.content.parts:
            for part in event.content.parts:
                if part.text:
                    response += part.text

    return response

async def run_visualization():
    """Run visualization agent if user asks for charts."""
    message = Content(
        role="user",
        parts=[Part.from_text(
            text=f"Create a visualization for: {prompt}\n{context}"
        )]
    )

    response = ""
    inline_data = []

    async for event in viz_runner.run_async(
        user_id="streamlit_user",
        session_id=st.session_state.viz_session_id,
        new_message=message
    ):
        if event.content and event.content.parts:
            for part in event.content.parts:
                if part.text:
                    response += part.text
                # Handle inline data (visualizations)
                if hasattr(part, 'inline_data') and part.inline_data:
                    inline_data.append(part.inline_data)
```

```
    return response, inline_data

# Detect visualization requests
if any(word in prompt.lower()
       for word in ['chart', 'plot', 'graph', 'visualiz', 'show']):
    response_text, viz_data = asyncio.run(run_visualization())

    # Display inline images
    for viz in viz_data:
        try:
            import base64
            from io import BytesIO
            from PIL import Image

            if hasattr(viz, 'data'):
                image_bytes = (
                    base64.b64decode(viz.data)
                    if isinstance(viz.data, str)
                    else viz.data
                )
                image = Image.open(BytesIO(image_bytes))
                st.image(image, use_column_width=True)
        except Exception as e:
            st.warning(f"Could not display viz: {str(e)}")
else:
    response_text = asyncio.run(run_analysis())
```

When to Use Multi-Agent Patterns

Scenario	Pattern	Benefit
Simple Q&A	Single agent	Fast, simple
Analysis + charts	Multi-agent	Better separation
Code generation	Agent + executor	Safe execution
Complex workflows	Pipeline agents	Scalable

Key Insight: Multi-agent systems let you:

- ✔️ Specialize agents by function

- ✔️ Reuse agents across projects
- ✔️ Execute code safely with executors
- ✔️ Handle complex workflows
- ✔️ Scale independent of frontend

Building a Data Analysis App

Feature 1: Interactive Visualizations

Add chart generation using Plotly:

```
import plotly.express as px

def create_chart(chart_type: str, column_x: str, column_y: str = None,
                 title: str = None) -> dict:
    """Create a visualization chart."""
    if st.session_state.df is None:
        return {"error": "No dataset loaded"}

    df = st.session_state.df

    try:
        if chart_type == "histogram":
            fig = px.histogram(
                df,
                x=column_x,
                title=title or f"Distribution of {column_x}"
            )

        elif chart_type == "scatter":
            fig = px.scatter(
                df,
                x=column_x,
                y=column_y,
                title=title or f"{column_y} vs {column_x}",
                trendline="ols"
            )

        elif chart_type == "bar":
            if column_y:
                data = df.groupby(column_x)[column_y].sum().reset_index()
                fig = px.bar(data, x=column_x, y=column_y,
                             title=title or f"{column_y} by {column_x}")
            else:
                fig = px.bar(df[column_x].value_counts().head(10),
                             title=title or f"Top 10 {column_x}")

        else:
            return {"error": "Unknown chart type"}

        st.session_state.last_chart = fig
        return {"success": True, "chart_type": chart_type}

    except Exception as e:
        return {"error": f"Chart error: {str(e)}"}
```

Usage:

```
# In your assistant response handler
if "show me a histogram" in prompt.lower():
    create_chart("histogram", "price")
    st.plotly_chart(st.session_state.last_chart)
```

Feature 2: Interactive Visualizations

Add chart generation:

```
def create_chart(chart_type: str, column_x: str, column_y: str = None, title:
    """
    Create a visualization chart.

    Args:
        chart_type: Type of chart (bar, line, scatter, histogram, box)
        column_x: Column for x-axis
        column_y: Column for y-axis (optional for histogram)
        title: Chart title

    Returns:
        Dict with chart data or error
    """
    if st.session_state.dataframe is None:
        return {"error": "No dataset loaded"}

    df = st.session_state.dataframe

    # Use filtered data if available
    if st.session_state.filtered_dataframe is not None:
        df = st.session_state.filtered_dataframe

    try:
        if chart_type == "histogram":
            if column_x not in df.columns:
                return {"error": f"Column '{column_x}' not found"}

            fig = px.histogram(
                df,
                x=column_x,
                title=title or f"Distribution of {column_x}"
            )

        elif chart_type == "bar":
            if column_x not in df.columns:
                return {"error": f"Column '{column_x}' not found"}

            # Aggregate data for bar chart
            if column_y:
                chart_data = df.groupby(column_x)[column_y].sum().reset_index(
                fig = px.bar(
                    chart_data,
                    x=column_x,
                    y=column_y,
                    title=title or f"{column_y} by {column_x}"
                )
```

```
            else:
                value_counts = df[column_x].value_counts().head(10)
                fig = px.bar(
                    x=value_counts.index,
                    y=value_counts.values,
                    title=title or f"Top 10 {column_x}",
                    labels={"x": column_x, "y": "Count"}
                )

        elif chart_type == "scatter":
            if not column_y:
                return {"error": "Scatter plot requires both x and y columns"}

            if column_x not in df.columns or column_y not in df.columns:
                return {"error": "Column not found"}

            fig = px.scatter(
                df,
                x=column_x,
                y=column_y,
                title=title or f"{column_y} vs {column_x}",
                trendline="ols"
            )

        elif chart_type == "box":
            if column_x not in df.columns:
                return {"error": f"Column '{column_x}' not found"}

            fig = px.box(
                df,
                y=column_x,
                title=title or f"Distribution of {column_x}"
            )

        elif chart_type == "line":
            if not column_y:
                return {"error": "Line plot requires both x and y columns"}

            if column_x not in df.columns or column_y not in df.columns:
                return {"error": "Column not found"}

            fig = px.line(
                df,
                x=column_x,
                y=column_y,
                title=title or f"{column_y} over {column_x}"
            )
```

```
        else:
            return {"error": "Unknown chart type"}

        # Store chart in session state for display
        st.session_state.last_chart = fig

        return {
            "success": True,
            "chart_type": chart_type,
            "description": f"Created {chart_type} chart with {len(df)} data po
        }

    except Exception as e:
        return {"error": f"Chart error: {str(e)}"}

# Add to agent tools
FunctionDeclaration(
    name="create_chart",
    description="Create a visualization chart from the dataset",
    parameters={
        "type": "object",
        "properties": {
            "chart_type": {
                "type": "string",
                "description": "Type of chart to create",
                "enum": ["bar", "line", "scatter", "histogram", "box"]
            },
            "column_x": {
                "type": "string",
                "description": "Column for x-axis"
            },
            "column_y": {
                "type": "string",
                "description": "Column for y-axis (optional for some chart typ
            },
            "title": {
                "type": "string",
                "description": "Chart title"
            }
        },
        "required": ["chart_type", "column_x"]
    }
)

# Update tools mapping
TOOLS = {
```

```
    "analyze_column": analyze_column,
    "calculate_correlation": calculate_correlation,
    "filter_data": filter_data,
    "create_chart": create_chart
}

# Display charts in chat
for message in st.session_state.messages:
    with st.chat_message(message["role"]):
        st.markdown(message["content"])

        # Check if chart should be displayed after this message
        if message["role"] == "assistant" and "last_chart" in st.session_state
            st.plotly_chart(st.session_state.last_chart, use_container_width=T
            # Clear chart after displaying
            del st.session_state.last_chart
```

Try it:

- "Create a histogram of the price column"
- "Show me a scatter plot of price vs sales"
- "Make a bar chart of revenue by category"

Beautiful charts appear inline! 📈

ADK Runner Integration with Streamlit

Now let's dive deep into how to properly integrate ADK Runners with Streamlit's execution model.

Understanding Session Management

ADK Runners need session management for stateful conversations:

```python
from google.adk.runners import Runner
from google.adk.sessions import InMemorySessionService
import asyncio
import uuid

# ===== SETUP =====

# Option 1: In-Memory Sessions (Development)
session_service = InMemorySessionService()

# Option 2: Persistent Sessions (Production)
# from google.cloud import firestore
# session_service = FirestoreSessionService(db)

runner = Runner(
    agent=root_agent,
    app_name="my_analysis_app",
    session_service=session_service
)

# ===== STREAMLIT INTEGRATION =====

@st.cache_resource
def get_runner_and_service():
    """Cache runner and session service."""
    session_service = InMemorySessionService()
    runner = Runner(
        agent=root_agent,
        app_name="data_analysis_assistant",
        session_service=session_service,
    )
    return runner, session_service

runner, session_service = get_runner_and_service()

# Initialize session on first load
if "adk_session_id" not in st.session_state:
    async def create_session():
        session = await session_service.create_session(
            app_name="data_analysis_assistant",
            user_id="streamlit_user"
        )
        return session.id

    st.session_state.adk_session_id = asyncio.run(create_session())
```

Async Execution Pattern

ADK agents are async-first. Here's how to properly run them in Streamlit:

```
from google.genai.types import Content, Part
import asyncio

async def run_agent_query(message_text: str) -> str:
    """Execute agent query and return response."""

    # Create structured message
    message = Content(
        role="user",
        parts=[Part.from_text(text=message_text)]
    )

    # Collect response
    response = ""

    # Execute agent
    async for event in runner.run_async(
        user_id="streamlit_user",
        session_id=st.session_state.adk_session_id,
        new_message=message
    ):
        # Handle streaming events
        if event.content and event.content.parts:
            for part in event.content.parts:
                # Handle text responses
                if part.text:
                    response += part.text

                # Handle inline data (images/charts)
                if hasattr(part, 'inline_data') and part.inline_data:
                    st.image(part.inline_data.data)

                # Handle code execution results
                if hasattr(part, 'code_execution_result'):
                    result = part.code_execution_result
                    if result.outcome == "SUCCESS":
                        st.success(f"Code executed: {result.output}")

    return response

# In your chat handler
if prompt := st.chat_input("Ask..."):
    with st.chat_message("assistant"):
        with st.status("Processing...", expanded=False) as status:
            try:
                # Run async agent
```

```
                response = asyncio.run(run_agent_query(prompt))
                status.update(label="✔️ Done", state="complete")
            except Exception as e:
                status.update(label="❌ Error", state="error")
                st.error(str(e))
                response = f"Error: {str(e)}"

        st.markdown(response)
```

Caching Best Practices

Optimize Streamlit + ADK performance:

```
import hashlib

# Cache agent execution results
@st.cache_data
def cached_agent_run(
    prompt: str,
    df_hash: str,
    _runner
) -> str:
    """Cache agent responses based on input hash."""
    return asyncio.run(run_agent_query(prompt))

# In your handler
if st.session_state.df is not None:
    # Create hash of dataframe (cache key)
    df_hash = hashlib.md5(
        st.session_state.df.to_json().encode()
    ).hexdigest()

    # Use cached execution
    response = cached_agent_run(
        prompt=prompt,
        df_hash=df_hash,
        _runner=runner  # Underscore prevents caching
    )
```

Error Handling

ADK Runner operations need robust error handling:

```python
from google.adk.runners import TimeoutError
from google.genai import APIError

async def run_agent_safely(message_text: str) -> tuple[str, bool]:
    """Run agent with error handling.

    Returns:
        (response_text, success: bool)
    """
    try:
        message = Content(
            role="user",
            parts=[Part.from_text(text=message_text)]
        )

        response = ""

        async for event in runner.run_async(
            user_id="streamlit_user",
            session_id=st.session_state.adk_session_id,
            new_message=message,
            timeout=30  # 30 second timeout
        ):
            if event.content and event.content.parts:
                for part in event.content.parts:
                    if part.text:
                        response += part.text

        return response, True

    except TimeoutError:
        return (
            "⏱️ Request timed out. Try a simpler query.",
            False
        )
    except APIError as e:
        return (
            f"❌ API Error: {str(e)}",
            False
        )
    except Exception as e:
        return (
            f"❌ Unexpected error: {type(e).__name__}: {str(e)}",
            False
        )
```

```
# Usage
response, success = asyncio.run(run_agent_safely(prompt))

if not success:
    st.error(response)
else:
    st.markdown(response)
```

State Persistence Patterns

Store conversation history correctly:

```
# Option 1: Streamlit Session State
# Persists within single browser session
if "messages" not in st.session_state:
    st.session_state.messages = []

st.session_state.messages.append({
    "role": "user",
    "content": prompt
})

# Option 2: Database Persistence
# Persists across sessions for a user
import json
from datetime import datetime

def save_to_database(message):
    """Save message to Firestore or similar."""
    db.collection("conversations").add({
        "user_id": "streamlit_user",
        "session_id": st.session_state.adk_session_id,
        "timestamp": datetime.now(),
        "message": message
    })

# Option 3: Multi-Session State
# Different conversation per tab
if "tab_sessions" not in st.session_state:
    st.session_state.tab_sessions = {}

active_tab = st.tabs(["Chat", "Analysis", "Visualizations"])[0]
if "current_tab" not in st.session_state:
    st.session_state.current_tab = "Chat"

# Create separate session per tab
tab_key = f"session_{st.session_state.current_tab}"
if tab_key not in st.session_state:
    session = asyncio.run(session_service.create_session(
        app_name="multi_tab_app",
        user_id="streamlit_user"
    ))
    st.session_state[tab_key] = session.id
```

Performance Optimization

Key optimization patterns:

```python
# 1. Use streaming for long responses
async def stream_agent_response():
    """Stream response chunks as they arrive."""
    message_placeholder = st.empty()
    full_response = ""

    async for event in runner.run_async(...):
        if event.content and event.content.parts:
            for part in event.content.parts:
                if part.text:
                    full_response += part.text
                    # Update UI in real-time
                    message_placeholder.markdown(
                        full_response + " ▌"  # Blinking cursor
                    )

    return full_response

# 2. Batch multiple queries
async def batch_queries(queries: list[str]) -> list[str]:
    """Execute multiple agent queries efficiently."""
    tasks = [
        run_agent_query(q)
        for q in queries
    ]
    return await asyncio.gather(*tasks)

# 3. Implement exponential backoff
async def run_with_retry(
    message: str,
    max_retries: int = 3
) -> str:
    """Run agent with automatic retries."""
    for attempt in range(max_retries):
        try:
            return asyncio.run(run_agent_query(message))
        except Exception as e:
            if attempt == max_retries - 1:
                raise

            wait_time = 2 ** attempt  # Exponential backoff
            st.warning(f"Retry in {wait_time}s...")
            await asyncio.sleep(wait_time)
```

Advanced Features

Feature 1: Multi-Dataset Support

Allow users to work with multiple datasets:

```
# Enhanced session state
if "datasets" not in st.session_state:
    st.session_state.datasets = {}

if "active_dataset" not in st.session_state:
    st.session_state.active_dataset = None

# Sidebar
with st.sidebar:
    st.header("📁 Datasets")

    # File uploader
    uploaded_file = st.file_uploader(
        "Upload CSV",
        type=["csv"],
        key="uploader"
    )

    if uploaded_file is not None:
        dataset_name = st.text_input(
            "Dataset name",
            value=uploaded_file.name.replace(".csv", "")
        )

        if st.button("Load Dataset"):
            try:
                df = pd.read_csv(uploaded_file)
                st.session_state.datasets[dataset_name] = df
                st.session_state.active_dataset = dataset_name
                st.success(f"✔️ Loaded '{dataset_name}'")
                st.rerun()
            except Exception as e:
                st.error(f"Error: {e}")

    # Dataset selector
    if st.session_state.datasets:
        st.subheader("Active Dataset")
        active = st.selectbox(
            "Select dataset",
            options=list(st.session_state.datasets.keys()),
            index=list(st.session_state.datasets.keys()).index(
                st.session_state.active_dataset
            ) if st.session_state.active_dataset else 0
        )
        st.session_state.active_dataset = active
```

```
        # Show info about active dataset
        df = st.session_state.datasets[active]
        st.write(f"**Rows:** {len(df)}")
        st.write(f"**Columns:** {len(df.columns)}")

        # Preview
        with st.expander("Preview"):
            st.dataframe(df.head(), use_container_width=True)

# Update tools to use active dataset
def get_active_dataframe():
    """Get the currently active dataset."""
    if st.session_state.active_dataset and st.session_state.active_dataset in
        return st.session_state.datasets[st.session_state.active_dataset]
    return None

# Update tool functions to use get_active_dataframe()
```

Feature 2: Export Analysis Results

Let users download analysis results:

```
import json
from datetime import datetime

# Add export button in sidebar
if st.session_state.messages:
    st.sidebar.markdown("---")
    st.sidebar.subheader("💾 Export")

    if st.sidebar.button("Export Conversation"):
        # Create export data
        export_data = {
            "timestamp": datetime.now().isoformat(),
            "dataset": st.session_state.active_dataset,
            "conversation": st.session_state.messages
        }

        # Convert to JSON
        json_str = json.dumps(export_data, indent=2)

        # Download button
        st.sidebar.download_button(
            label="Download JSON",
            data=json_str,
            file_name=f"analysis_{datetime.now().strftime('%Y%m%d_%H%M%S')}.js
            mime="application/json"
        )

    # Export filtered data
    if st.session_state.filtered_dataframe is not None:
        if st.sidebar.button("Export Filtered Data"):
            csv = st.session_state.filtered_dataframe.to_csv(index=False)

            st.sidebar.download_button(
                label="Download CSV",
                data=csv,
                file_name=f"filtered_data_{datetime.now().strftime('%Y%m%d_%H%
                mime="text/csv"
            )
```

Feature 3: Caching for Performance

Optimize with Streamlit caching:

```python
# Cache expensive computations
@st.cache_data
def load_dataset(file):
    """Load and cache dataset."""
    return pd.read_csv(file)

@st.cache_data
def compute_statistics(df_hash, column_name):
    """Cache column statistics."""
    # df_hash is used as cache key
    df = st.session_state.dataframe
    return df[column_name].describe().to_dict()

# Cache visualizations
@st.cache_data
def create_cached_chart(chart_type, column_x, column_y, data_hash):
    """Cache chart generation."""
    df = st.session_state.dataframe
    # ... create chart
    return fig

# Use in tools
def analyze_column(column_name, analysis_type):
    df = st.session_state.dataframe

    # Use cached computation
    df_hash = hash(df.to_json())  # Simple hash for caching
    stats = compute_statistics(df_hash, column_name)

    return stats
```

This makes repeated queries blazing fast! ⚡

Production Deployment

Option 1: Streamlit Cloud (Easiest)

Step 1: Prepare Repository

```
# Create requirements.txt
cat > requirements.txt << EOF
streamlit==1.39.0
google-genai==1.41.0
pandas==2.2.0
plotly==5.24.0
EOF

# Create .streamlit/config.toml for better UX
mkdir .streamlit
cat > .streamlit/config.toml << EOF
[theme]
primaryColor = "#FF4B4B"
backgroundColor = "#FFFFFF"
secondaryBackgroundColor = "#F0F2F6"
textColor = "#262730"
font = "sans serif"

[server]
maxUploadSize = 200
EOF

# Create .streamlit/secrets.toml for API key
cat > .streamlit/secrets.toml << EOF
GOOGLE_API_KEY = "your_api_key_here"
EOF

# Add to .gitignore
echo ".streamlit/secrets.toml" >> .gitignore
```

Update `app.py` to use secrets:

Step 2: Deploy

```
**Update `app.py` to use secrets**:

```python
import os
import streamlit as st

# Get API key from secrets or environment
api_key = st.secrets.get("GOOGLE_API_KEY") or os.getenv("GOOGLE_API_KEY")

if not api_key:
    st.error("Please configure GOOGLE_API_KEY in Streamlit secrets")
    st.stop()

client = genai.Client(
    api_key=api_key,
    http_options={'api_version': 'v1alpha'}
)
```

Step 2: Deploy (Streamlit Cloud)

1. Push code to GitHub
2. Go to share.streamlit.io (https://share.streamlit.io)
3. Click "New app"
4. Select your repository
5. Set main file: `app.py`
6. Add secret: `GOOGLE_API_KEY = your_key`
7. Click "Deploy"!

Your app is live! 🎉

URL: `https://your-app.streamlit.app`

Option 2: Google Cloud Run

For more control and custom domains:
```

Step 1: Create Dockerfile

```
FROM python:3.11-slim

WORKDIR /app

# Install dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy app
COPY app.py .
COPY .streamlit/ .streamlit/

# Expose Streamlit port
EXPOSE 8501

# Health check
HEALTHCHECK CMD curl --fail http://localhost:8501/_stcore/health || exit 1

# Run app
CMD ["streamlit", "run", "app.py", "--server.port=8501", "--server.address=0.0
```

Step 2: Deploy (Cloud Run)

```
# Build and deploy
gcloud run deploy data-analysis-agent \
  --source=. \
  --region=us-central1 \
  --allow-unauthenticated \
  --set-env-vars="GOOGLE_API_KEY=your_api_key" \
  --port=8501

# Output:
# Service URL: https://data-analysis-agent-abc123.run.app
```

Step 3: Custom Domain (Optional)

```
# Map custom domain
gcloud run domain-mappings create \
  --service=data-analysis-agent \
  --domain=analyze.yourdomain.com \
  --region=us-central1
```

Production Best Practices

1. Rate Limiting

```
import time
from collections import defaultdict

# Simple rate limiter
class RateLimiter:
    def __init__(self, max_requests=10, window=60):
        self.max_requests = max_requests
        self.window = window
        self.requests = defaultdict(list)

    def is_allowed(self, user_id):
        now = time.time()
        # Clean old requests
        self.requests[user_id] = [
            req_time for req_time in self.requests[user_id]
            if now - req_time < self.window
        ]

        if len(self.requests[user_id]) < self.max_requests:
            self.requests[user_id].append(now)
            return True
        return False

# Use in app
rate_limiter = RateLimiter(max_requests=20, window=60)

if prompt := st.chat_input("Ask me..."):
    # Simple user ID (use actual auth in production)
    user_id = st.session_state.get("session_id", "default")

    if not rate_limiter.is_allowed(user_id):
        st.error("Too many requests. Please wait a minute.")
        st.stop()

    # ... process request
```

2. Error Handling

```
import logging

# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)

# Wrap agent calls
try:
    # Proper ADK execution pattern with InMemoryRunner
    import asyncio
    from google.genai import types

    async def get_response(message: str):
        """Helper to execute agent in async context."""
        new_message = types.Content(role='user', parts=[types.Part(text=messag

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

        return response_text

    response = asyncio.run(get_response(message))
    # ... process response
except Exception as e:
    logger.error(f"Agent error: {e}", exc_info=True)
    st.error("I encountered an error. Our team has been notified.")

    # Don't expose internal errors to users
    if os.getenv("ENVIRONMENT") == "development":
        st.exception(e)
```

3. Monitoring

```
from google.cloud import monitoring_v3
import time

def log_metric(metric_name, value):
    """Log metric to Cloud Monitoring."""
    if os.getenv("ENVIRONMENT") != "production":
        return

    client = monitoring_v3.MetricServiceClient()
    project_name = f"projects/{os.getenv('GCP_PROJECT')}"

    series = monitoring_v3.TimeSeries()
    series.metric.type = f"custom.googleapis.com/{metric_name}"

    now = time.time()
    seconds = int(now)
    nanos = int((now - seconds) * 10 ** 9)
    interval = monitoring_v3.TimeInterval(
        {"end_time": {"seconds": seconds, "nanos": nanos}}
    )
    point = monitoring_v3.Point(
        {"interval": interval, "value": {"double_value": value}}
    )
    series.points = [point]

    client.create_time_series(name=project_name, time_series=[series])

# Use in app
start_time = time.time()

# Proper ADK execution pattern
import asyncio
from google.genai import types

async def get_response(message: str):
    """Helper to execute agent in async context."""
    new_message = types.Content(role='user', parts=[types.Part(text=message)])

    response_text = ""
    async for event in runner.run_async(
        user_id=st.session_state.get("user_id", "streamlit_user"),
        session_id=st.session_state.session_id,
        new_message=new_message
    ):
        if event.content and event.content.parts:
            response_text += event.content.parts[0].text
```

```
    return response_text

response = asyncio.run(get_response(message))

latency = time.time() - start_time

log_metric("agent_latency", latency)
log_metric("agent_requests", 1)
```

4. Session Management

```
import uuid

# Generate unique session ID
if "session_id" not in st.session_state:
    st.session_state.session_id = str(uuid.uuid4())

# Store sessions in database (example with Firestore)
from google.cloud import firestore

db = firestore.Client()

def save_session():
    """Save session to Firestore."""
    doc_ref = db.collection("sessions").document(st.session_state.session_id)
    doc_ref.set({
        "messages": st.session_state.messages,
        "timestamp": firestore.SERVER_TIMESTAMP,
        "dataset": st.session_state.active_dataset
    })

def load_session(session_id):
    """Load session from Firestore."""
    doc_ref = db.collection("sessions").document(session_id)
    doc = doc_ref.get()

    if doc.exists:
        data = doc.to_dict()
        st.session_state.messages = data.get("messages", [])
        st.session_state.active_dataset = data.get("dataset")

# Auto-save on changes
if st.session_state.messages:
    save_session()
```

Troubleshooting

Common Issues

Issue 1: "Please set GOOGLE_API_KEY"

Solution:

```
# Local development

streamlit run app.py

# Or create .streamlit/secrets.toml
echo 'GOOGLE_API_KEY = "your_key"' > .streamlit/secrets.toml
```

Issue 2: File Upload Not Working

Symptoms:

- Upload button doesn't respond
- File shows but data doesn't load

Solution:

```
# Check file encoding
uploaded_file = st.file_uploader("Upload CSV", type=["csv"])

if uploaded_file is not None:
    try:
        # Try UTF-8 first
        df = pd.read_csv(uploaded_file, encoding='utf-8')
    except UnicodeDecodeError:
        # Fallback to latin-1
        df = pd.read_csv(uploaded_file, encoding='latin-1')
    except Exception as e:
        st.error(f"Error loading file: {e}")
        st.stop()
```

Issue 3: Agent Not Using Tools

Symptoms:

- Agent responds generically
- No function calls executed

Solution:

```
from google.adk.agents import Agent

# Verify tool registration
agent = Agent(
    model="gemini-2.0-flash-exp",
    name="data_analysis_agent",
    instruction="...",
    tools=[analyze_column, calculate_correlation, filter_data, get_dataset_sum
)

# ADK automatically handles function calling configuration
# Tools are enabled by default in AUTO mode

# Check tool names match function names
TOOLS = {
    "analyze_column": analyze_column,  # ✔️ Function name matches
    "analyzeColumn": analyze_column,   # ❌ Wrong name
}
```

Issue 4: Slow Chart Generation

Symptoms:

- Charts take 5+ seconds to load
- App feels laggy

Solution:

```python
# Use caching
@st.cache_data
def create_cached_chart(chart_type, x_col, y_col, data_hash):
    """Cache expensive chart operations."""
    df = st.session_state.dataframe

    if chart_type == "scatter":
        # Sample large datasets
        if len(df) > 10000:
            df = df.sample(n=10000)

    fig = px.scatter(df, x=x_col, y=y_col)
    return fig

# Use hash for cache key
df_hash = hash(df.to_json())  # Or use df.shape + df.columns
fig = create_cached_chart("scatter", "x", "y", df_hash)
st.plotly_chart(fig)
```

Issue 5: Session State Lost on Refresh

Symptoms:

- Conversation disappears on page refresh
- Uploaded data is lost

Solution:

```python
# Option 1: Use query params for session ID
import streamlit as st

# Get session ID from URL
query_params = st.query_params
session_id = query_params.get("session", str(uuid.uuid4()))

# Set in URL
st.query_params["session"] = session_id

# Load from database
load_session(session_id)

# Option 2: Use cookies (requires streamlit-cookies)
# pip install streamlit-cookies-manager
from streamlit_cookies_manager import EncryptedCookieManager

cookies = EncryptedCookieManager(
    prefix="myapp",
    password=os.environ["COOKIE_PASSWORD"]
)

if not cookies.ready():
    st.stop()

# Store session ID in cookie
if "session_id" not in cookies:
    cookies["session_id"] = str(uuid.uuid4())
    cookies.save()

session_id = cookies["session_id"]
```

Next Steps

You've Mastered Streamlit + ADK! 🎉

You now know how to:

✔️ Build pure Python data apps with ADK
✔️ Integrate agents directly (no HTTP overhead!)
✔️ Create interactive chat interfaces with Streamlit
✔️ Add data analysis tools and visualizations
✔️ Deploy to Streamlit Cloud and Cloud Run
✔️ Optimize with caching and error handling

Compare Integration Approaches

Feature	Streamlit	Next.js	React Vite
Language	Python only	TypeScript + Python	TypeScript + Python
Setup Time	<5 min	~15 min	~10 min
Architecture	In-process	HTTP	HTTP
Latency	~0ms	~50ms	~50ms
Customization	Medium	High	High
Data Tools	Excellent	Good	Good
Best For	Data apps	Web apps	Lightweight apps

Continue Learning

Tutorial 33: Slack Bot Integration with ADK
Build a team support bot that works in Slack channels

Tutorial 34: Google Cloud Pub/Sub + Event-Driven Agents
Build scalable event-driven agent architectures

Tutorial 35: AG-UI Deep Dive
Master advanced CopilotKit features for enterprise apps

Additional Resources

- Streamlit Documentation (https://docs.streamlit.io)
- ADK Documentation (https://google.github.io/adk-docs/)

- Streamlit Gallery (https://streamlit.io/gallery) - Inspiration
- Streamlit Components (https://streamlit.io/components) - Extensions

---

🎉 Tutorial 32 Complete!

Next: Tutorial 33: Slack Bot Integration (./33_slack_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 09:03:06 from 32_streamlit_adk_integration.md

Source: Google ADK Training Hub