Tutorial 14: Streaming and Server-Sent Events (SSE) - Real-Time Responses

Difficulty: advanced
Reading Time: 1.5 hours
Tags: advanced, streaming, sse, real-time, chat
Description: Build streaming agents that deliver real-time responses to users, perfect for chat interfaces and live updates using Server-Sent Events.

:::info IMPLEMENTATION NOTES

**This tutorial demonstrates real ADK streaming APIs with a working implementation
that uses ADK v1.16.0's actual streaming capabilities.**

- ADK v1.16.0 includes full streaming support with `StreamingMode`, `Runner`, `Session`, and `LiveRequestQueue` classes
- The tutorial implementation uses actual ADK streaming APIs, not simulation
- `Runner.run_async()` with `StreamingMode.SSE` provides real progressive output
- `Session` management maintains conversation context across streaming responses
- `LiveRequestQueue` enables bidirectional communication for advanced streaming scenarios

Working Implementation Available:
Check out the working implementation (https://github.com/raphaelmansuy/adk_training/tree/main/tutorial_implementation/tutorial14)
which demonstrates real ADK streaming APIs with proper session management and error handling.

:::

Tutorial 14: Streaming with Server-Sent Events (SSE)

Goal: Implement streaming responses using Server-Sent Events (SSE) to provide real-time, progressive output for better user experience in your AI agents.

Prerequisites:

- Tutorial 01 (Hello World Agent)
- Tutorial 02 (Function Tools)
- Basic understanding of async/await in Python

What You'll Learn:

- Understanding streaming vs. non-streaming responses
- Implementing SSE streaming with `StreamingMode.SSE`
- Using `RunConfig` for streaming configuration
- Building real-time chat interfaces
- Aggregating streaming responses
- Best practices for production streaming applications

Time to Complete: 45-60 minutes

Why Streaming Matters

Traditional AI responses are **blocking** - users wait for the complete answer before seeing anything. Streaming provides **progressive output** as the model generates text.

Without Streaming (Blocking):

```
User: "Explain quantum computing"
Agent: [5 seconds of waiting...]
       [Complete response appears at once]
```

With Streaming (Progressive):

```
User: "Explain quantum computing"
Agent: "Quantum computing is a revolutionary..."
       [Text appears word-by-word or chunk-by-chunk]
       [User sees progress immediately]
```

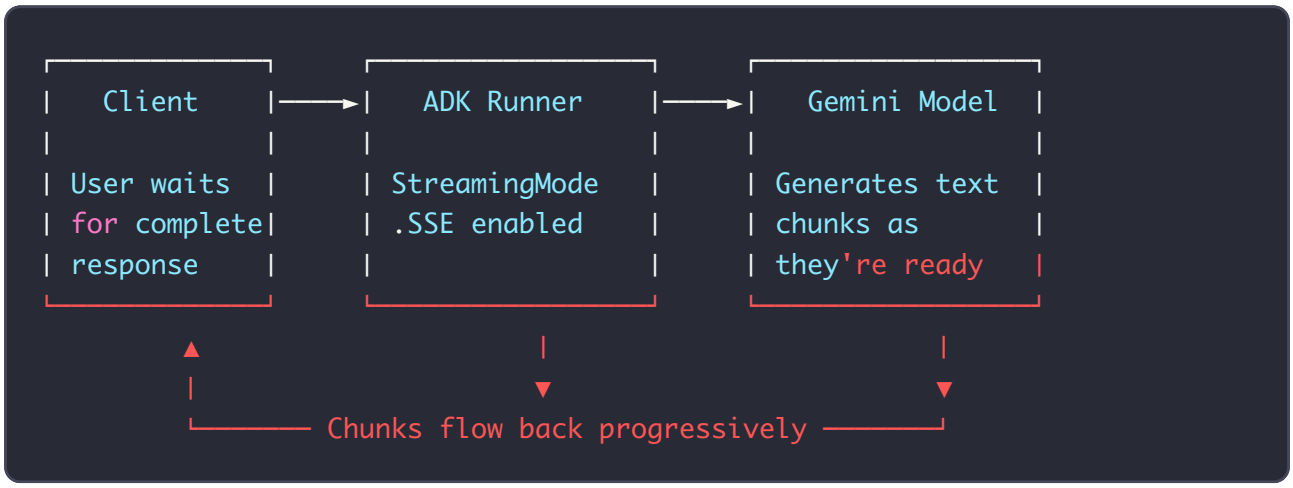
Benefits:

- ✔️ **Better UX**: Users see progress immediately
- ✔️ **Perceived Speed**: Feels faster even if total time is similar
- ✔️ **Early Feedback**: Users can interrupt if needed
- ✔️ **Real-Time Feel**: More conversational and engaging
- ✔️ **Long Responses**: Essential for lengthy outputs

Response Flow Comparison:

```
BLOCKING RESPONSE (Traditional):
User ──► [Agent Processing: 5 seconds] ──► Complete Response Displayed

STREAMING RESPONSE (Progressive):
User ──► Agent: "Quantum computing..." ──► "is revolutionary..." ──► "...appro
         [Immediate feedback]              [Progressive chunks]        [Conti
```

Data Flow Architecture:



1. Streaming Basics

What is Server-Sent Events (SSE)?

SSE is a standard protocol for servers to push data to clients over HTTP. In ADK, streaming enables the model to send response chunks as they're generated using `StreamingMode.SSE`.

Basic Streaming Implementation

```python
import asyncio
from google.adk.agents import Agent
from google.adk.runners import Runner
from google.adk.agents.run_config import RunConfig, StreamingMode
from google.adk.sessions import InMemorySessionService
from google.genai import types

# Create agent
agent = Agent(
    model='gemini-2.0-flash',
    name='streaming_assistant',
    instruction='Provide detailed, helpful responses.'
)

# Configure streaming
run_config = RunConfig(
    streaming_mode=StreamingMode.SSE
)

async def stream_response(query: str):
    """Stream agent response using real ADK APIs."""
    # Create session service and runner
    session_service = InMemorySessionService()
    runner = Runner(app_name="streaming_demo", agent=agent, session_service=se

    # Create session
    session = await session_service.create_session(
        app_name="streaming_demo",
        user_id="demo_user"
    )

    print(f"User: {query}\n")
    print("Agent: ", end='', flush=True)

    # Run with streaming
    async for event in runner.run_async(
        user_id="demo_user",
        session_id=session.id,
        new_message=types.Content(role="user", parts=[types.Part(text=query)])
        run_config=run_config
    ):
        # Print each chunk as it arrives
        if event.content and event.content.parts:
            for part in event.content.parts:
                if part.text:
                    print(part.text, end='', flush=True)
```

```
    print("\n")

# Usage
asyncio.run(stream_response("Explain how neural networks work"))
```

Output (progressive):

```
User: Explain how neural networks work

Agent: Neural networks are computational models inspired by...
       [Text appears progressively as generated]
       ...the human brain. They consist of interconnected nodes...
       [Continues streaming...]
       ...making them powerful for pattern recognition tasks.
```

How Streaming Works (Actual Implementation)

Current ADK v1.16.0 Implementation Flow:

1. **Setup Components** → Create `Runner`, `SessionService`, and `Session` for context
2. **Configure Streaming** → Use `RunConfig` with `StreamingMode.SSE`
3. **Send Message** → Use `types.Content` with proper role and parts structure
4. **Process Events** → Iterate through `runner.run_async()` events
5. **Extract Chunks** → Get text from `event.content.parts`
6. **Display Progressively** → Yield/print chunks as they arrive
7. **Complete** → Final event signals completion

Key Components:

- `Runner`: Executes agent runs with streaming support
- `SessionService`: Manages conversation sessions and context
- `RunConfig`: Configures streaming mode and parameters
- `StreamingMode.SSE`: Enables Server-Sent Events streaming
- `types.Content`: Properly structured message format

ADK Streaming Flow:


Initialize Components
Create Session Service & Runner
Configure StreamingMode .SSE
Send User Message (types.Content)
Process Events (Async Loop)
Extract Text Chunks from event.content
Display Progressively (flush=True)
Final Event Signals End
Complete Response

2. StreamingMode Configuration

Available Streaming Modes

```
from google.adk.agents import StreamingMode

# SSE - Server-Sent Events (one-way streaming)
StreamingMode.SSE

# BIDI - Bidirectional streaming (two-way, for Live API)
StreamingMode.BIDI

# OFF - No streaming (default, blocking)
StreamingMode.NONE
```

RunConfig Setup

```
from google.adk.agents import RunConfig, StreamingMode

# SSE Streaming
sse_config = RunConfig(
    streaming_mode=StreamingMode.SSE
)

# No Streaming (blocking)
blocking_config = RunConfig(
    streaming_mode=StreamingMode.NONE
)

# Use in runner
runner = Runner()

# Streaming
async for event in runner.run_async(query, agent, run_config=sse_config):
    process_event(event)

# Blocking
result = await runner.run_async(query, agent, run_config=blocking_config)
process_complete_result(result)
```

StreamingMode Decision Tree:


StreamingMode Selection Guide
What type of streaming?
SSE
(One-way
Server
Push)
BIDI
(Two-way
Live
API)
Use Cases:
• Chat apps
• Text streaming
• Progressive responses
• Web APIs
Use Cases:
• Voice agents
• Real-time audio/video
• Interactive conversations
• Live sessions
Data Flow Patterns:
SSE (Server-Sent Events):
Client ← Text Chunks — Server
← Text Chunks —
← Text Chunks —
BIDI (Bidirectional):
Client — Audio/Text → Server
← Audio/Text —
— Audio/Text →
← Audio/Text —
NONE (Blocking):
Client — Request → Server
← Full Response — (after complete processing)

3. Real-World Example: Streaming Chat Application

Complete Implementation

```
"""
Streaming Chat Application with SSE
Real-time interactive chat with progressive responses.
"""

import asyncio
import os
from datetime import datetime
from typing import AsyncIterator
from google.adk.agents import Agent, Runner, RunConfig, StreamingMode, Session
from google.genai import types

# Environment setup
os.environ['GOOGLE_GENAI_USE_VERTEXAI'] = '1'
os.environ['GOOGLE_CLOUD_PROJECT'] = 'your-project-id'
os.environ['GOOGLE_CLOUD_LOCATION'] = 'us-central1'

class StreamingChatApp:
    """Interactive streaming chat application."""

    def __init__(self):
        """Initialize chat application."""

        # Create chat agent
        self.agent = Agent(
            model='gemini-2.0-flash',
            name='chat_assistant',
            description='An assistant that can search the web.',
            instruction="""
You are a helpful, friendly assistant engaging in natural conversation.

Guidelines:
- Be conversational and engaging
- Provide detailed explanations when asked
- Ask clarifying questions if needed
- Remember conversation context
- Be concise for simple queries, detailed for complex ones
            """.strip(),
            generate_content_config=types.GenerateContentConfig(
                temperature=0.7,  # Conversational
                max_output_tokens=2048
            )
        )

        # Create session for conversation context
        self.session = Session()
```

```
        # Configure streaming
        self.run_config = RunConfig(
            streaming_mode=StreamingMode.SSE
        )

        self.runner = Runner()

    async def stream_response(self, user_message: str) -> AsyncIterator[str]:
        """
        Stream agent response to user message.

        Args:
            user_message: User's input message

        Yields:
            Text chunks as they're generated
        """

        # Run agent with streaming
        async for event in self.runner.run_async(
            user_message,
            agent=self.agent,
            session=self.session,
            run_config=self.run_config
        ):
            # Extract text from event
            if event.content and event.content.parts:
                for part in event.content.parts:
                    if part.text:
                        yield part.text

    async def chat_turn(self, user_message: str):
        """
        Execute one chat turn with streaming display.

        Args:
            user_message: User's input message
        """

        # Display user message
        timestamp = datetime.now().strftime('%H:%M:%S')
        print(f"\n[{timestamp}] User: {user_message}")

        # Display agent response with streaming
        timestamp = datetime.now().strftime('%H:%M:%S')
        print(f"[{timestamp}] Agent: ", end='', flush=True)
```

```
        # Stream response chunks
        async for chunk in self.stream_response(user_message):
            print(chunk, end='', flush=True)

        print()  # New line after complete response

    async def run_interactive(self):
        """Run interactive chat loop."""

        print("="*70)
        print("STREAMING CHAT APPLICATION")
        print("="*70)
        print("Type 'exit' or 'quit' to end conversation")
        print("="*70)

        while True:
            try:
                # Get user input
                user_input = input("\nYou: ").strip()

                if not user_input:
                    continue

                # Check for exit
                if user_input.lower() in ['exit', 'quit']:
                    print("\nGoodbye!")
                    break

                # Process chat turn
                await self.chat_turn(user_input)

            except KeyboardInterrupt:
                print("\n\nInterrupted. Goodbye!")
                break
            except Exception as e:
                print(f"\nError: {e}")

    async def run_demo(self):
        """Run demo conversation."""

        print("="*70)
        print("STREAMING CHAT DEMO")
        print("="*70)

        demo_messages = [
            "Hello! What can you help me with?",
```

```
            "Explain quantum computing in simple terms",
            "What are the practical applications?",
            "How does it compare to classical computing?"
        ]

        for message in demo_messages:
            await self.chat_turn(message)
            await asyncio.sleep(1)  # Pause between turns

async def main():
    """Main entry point."""

    chat = StreamingChatApp()

    # Run demo
    await chat.run_demo()

    # Uncomment for interactive mode:
    # await chat.run_interactive()

if __name__ == '__main__':
    asyncio.run(main())
```

Streaming Chat Application Architecture:


StreamingChatApp Class
Agent
(Gemini-2.0)
• Model config
• Instructions
• Temperature
Session
(Context)
• Conversation
history
• User state
Runner
(Executes)
• run_async
• Streaming
mode
RunConfig(streaming_mode=SSE)
Streaming Response Flow
User Input → Runner.run_async() → Events →
Chunks yielded progressively
chat_turn()
method
• Format msg
• Timestamp
• Error handle
stream_response
() async gen
• Yield chunks
• Handle async
• Context
Display
(flush=True)
• Progressive
output
• Real-time

Expected Output

```
================================================================================
STREAMING CHAT DEMO
================================================================================

[14:23:15] User: Hello! What can you help me with?
[14:23:15] Agent: Hello! I'm here to help with a wide variety of tasks...
[Streams progressively...]
...I can explain concepts, answer questions, help with writing, assist
with problem-solving, provide information on various topics, and much more.
What would you like to explore today?

[14:23:18] User: Explain quantum computing in simple terms
[14:23:18] Agent: Imagine regular computers use bits, which are like...
[Streams progressively...]
...light switches that are either ON (1) or OFF (0). Quantum computers use
quantum bits, or "qubits," which can be both ON and OFF at the same time...
[Continues streaming...]
...This allows quantum computers to explore many possibilities simultaneously,
making them potentially much faster for certain types of problems.

[14:23:25] User: What are the practical applications?
[14:23:25] Agent: Great question! Here are some key applications...
[Streams progressively...]

1. **Drug Discovery**: Simulating molecular interactions...
2. **Cryptography**: Breaking current encryption and creating quantum-safe...
3. **Optimization**: Solving complex logistics and scheduling...
4. **Financial Modeling**: Analyzing risk and portfolio optimization...
5. **Artificial Intelligence**: Training more sophisticated ML models...

[14:23:32] User: How does it compare to classical computing?
[14:23:32] Agent: Let me break down the key differences...
[Streams progressively...]

**Classical Computing:**
- Sequential processing (one calculation at a time)
- Deterministic (same input → same output)
- Excellent for everyday tasks...

**Quantum Computing:**
- Parallel exploration (many paths simultaneously)
- Probabilistic (results have probabilities)
- Excels at specific complex problems...

Think of it this way: a classical computer is like checking...
```

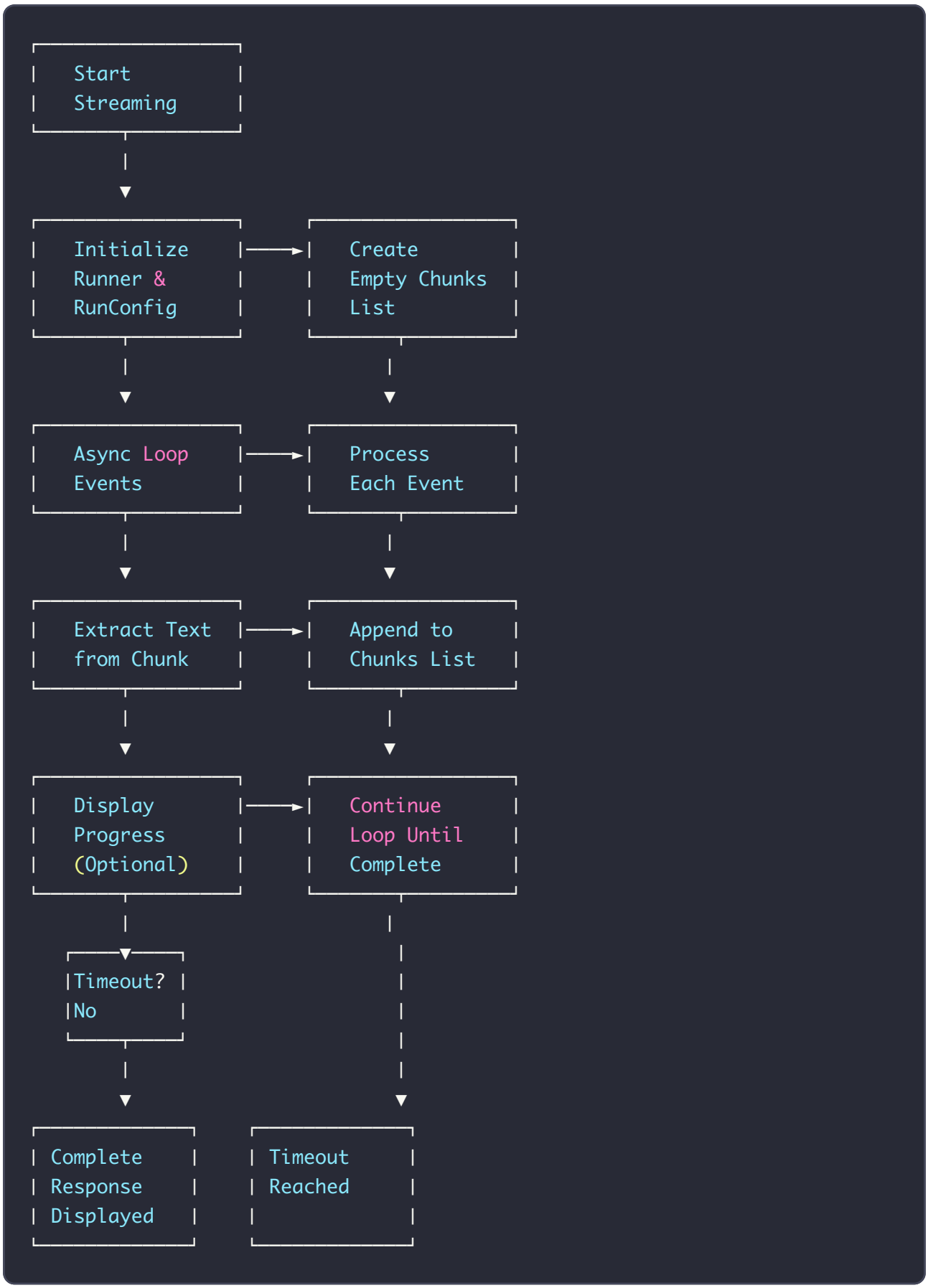
4. Advanced Streaming Patterns

Pattern 1: Response Aggregation

Collect the complete response while streaming:

```
from typing import List

async def stream_and_aggregate(query: str, agent: Agent) -> tuple[str, List[st
    """
    Stream response while collecting chunks.

    Returns:
        (complete_text, chunks_list)
    """

    runner = Runner()
    run_config = RunConfig(streaming_mode=StreamingMode.SSE)

    chunks = []

    async for event in runner.run_async(query, agent=agent, run_config=run_con
        if event.content and event.content.parts:
            chunk = event.content.parts[0].text
            chunks.append(chunk)
            print(chunk, end='', flush=True)

    complete_text = ''.join(chunks)
    return complete_text, chunks

# Usage
complete, chunks = await stream_and_aggregate(
    "Explain machine learning",
    agent
)

print(f"\n\nTotal chunks: {len(chunks)}")
print(f"Total length: {len(complete)} characters")
```

Pattern 1: Response Aggregation Flow:


Start Streaming
Initialize Runner & RunConfig
Create Empty Chunks List
Async Loop Events
Process Each Event
Extract Text from Chunk
Append to Chunks List
Display Progress (Optional)
Continue Loop Until Complete
Timeout? No
Complete Response Displayed
Timeout Reached

Pattern 2: Streaming with Progress Indicators

Show progress during streaming:

```
import sys

async def stream_with_progress(query: str, agent: Agent):
    """Stream with visual progress indicator."""

    runner = Runner()
    run_config = RunConfig(streaming_mode=StreamingMode.SSE)

    print("Agent: ", end='', flush=True)

    chunk_count = 0

    async for event in runner.run_async(query, agent=agent, run_config=run_con
        if event.content and event.content.parts:
            chunk = event.content.parts[0].text
            print(chunk, end='', flush=True)

            chunk_count += 1

            # Show progress indicator every 10 chunks
            if chunk_count % 10 == 0:
                sys.stderr.write('.')
                sys.stderr.flush()

    print()  # New line

# Usage
await stream_with_progress("Write a long essay on AI", agent)
```

Pattern 3: Streaming to Multiple Outputs

Send streaming response to multiple destinations:

```python
from typing import List, Callable

async def stream_to_multiple(
    query: str,
    agent: Agent,
    outputs: List[Callable[[str], None]]
):
    """
    Stream response to multiple output handlers.

    Args:
        query: User query
        agent: Agent to use
        outputs: List of functions to handle each chunk
    """

    runner = Runner()
    run_config = RunConfig(streaming_mode=StreamingMode.SSE)

    async for event in runner.run_async(query, agent=agent, run_config=run_con
        if event.content and event.content.parts:
            chunk = event.content.parts[0].text

            # Send to all outputs
            for output_fn in outputs:
                output_fn(chunk)

# Usage
async def console_output(chunk: str):
    print(chunk, end='', flush=True)

async def file_output(chunk: str):
    with open('response.txt', 'a') as f:
        f.write(chunk)

async def websocket_output(chunk: str):
    # await websocket.send(chunk)
    pass

await stream_to_multiple(
    "Explain AI safety",
    agent,
    outputs=[console_output, file_output, websocket_output]
)
```

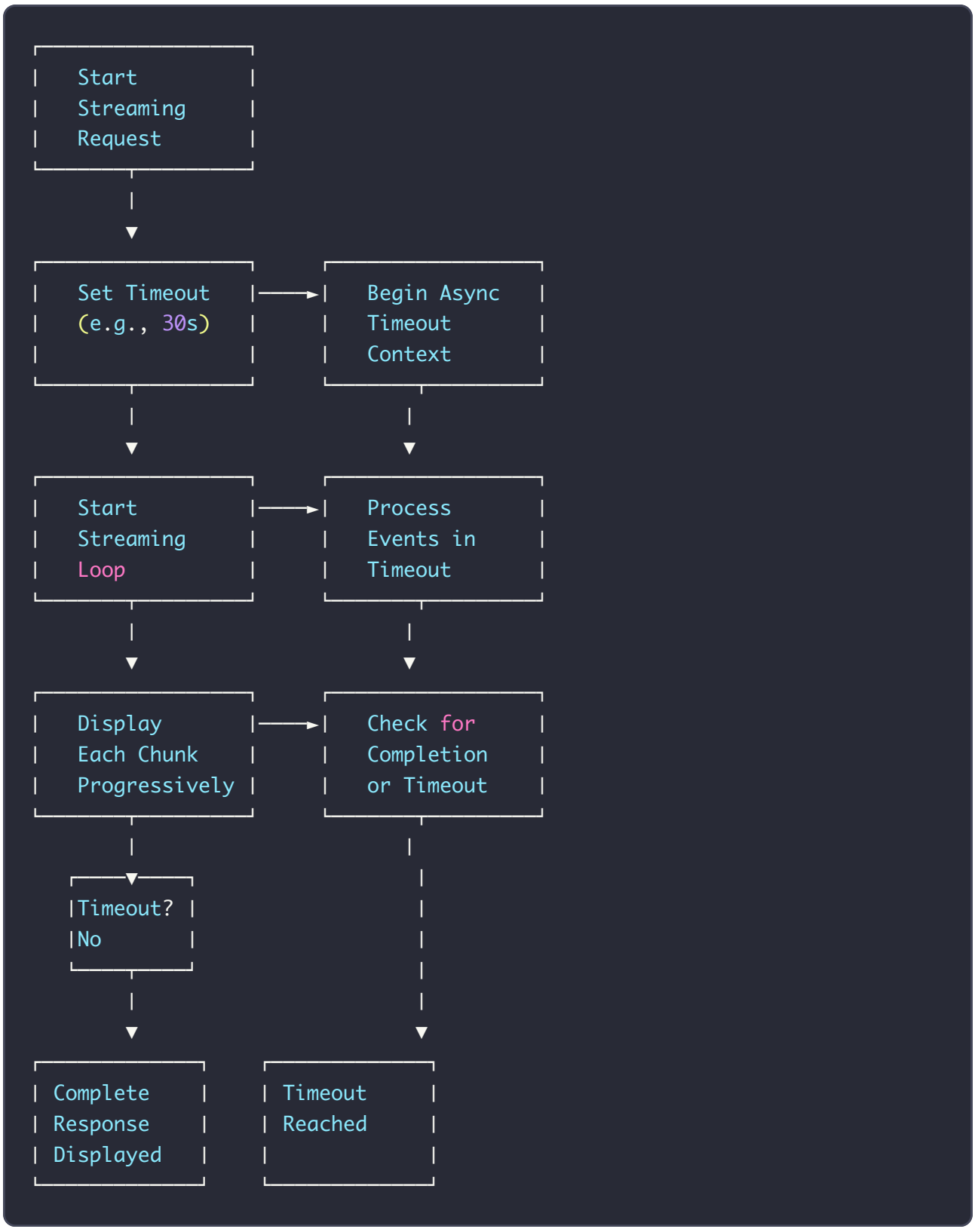
Pattern 3: Multiple Output Destinations:

Pattern 4: Streaming with Timeout

Add timeout protection:

```
import asyncio

async def stream_with_timeout(
    query: str,
    agent: Agent,
    timeout_seconds: float = 30.0
):
    """Stream response with timeout."""

    runner = Runner()
    run_config = RunConfig(streaming_mode=StreamingMode.SSE)

    try:
        async with asyncio.timeout(timeout_seconds):
            async for event in runner.run_async(query, agent=agent, run_config
                if event.content and event.content.parts:
                    chunk = event.content.parts[0].text
                    print(chunk, end='', flush=True)
    except asyncio.TimeoutError:
        print("\n\n[Timeout: Response took too long]")

    print()

# Usage
await stream_with_timeout("Explain the universe", agent, timeout_seconds=10.0)
```

Pattern 4: Timeout Protection Flow:


Start Streaming Request
Set Timeout (e.g., 30s)
Begin Async Timeout Context
Start Streaming Loop
Process Events in Timeout
Display Each Chunk Progressively
Check for Completion or Timeout
Timeout? No
Complete Response Displayed
Timeout Reached

5. StreamingResponseAggregator

ADK provides `StreamingResponseAggregator` for handling streaming responses:

```
from google.adk.models.streaming_response_aggregator import StreamingResponseA

async def stream_with_aggregator(query: str, agent: Agent):
    """Use StreamingResponseAggregator for cleaner code."""

    runner = Runner()
    run_config = RunConfig(streaming_mode=StreamingMode.SSE)

    aggregator = StreamingResponseAggregator()

    async for event in runner.run_async(query, agent=agent, run_config=run_con
        # Aggregator handles chunk collection
        aggregator.add(event)

        # Display chunk
        if event.content and event.content.parts:
            print(event.content.parts[0].text, end='', flush=True)

    # Get complete response
    complete_response = aggregator.get_response()

    print(f"\n\nComplete response has {len(complete_response.content.parts[0].

    return complete_response

# Usage
response = await stream_with_aggregator("Explain blockchain", agent)
```

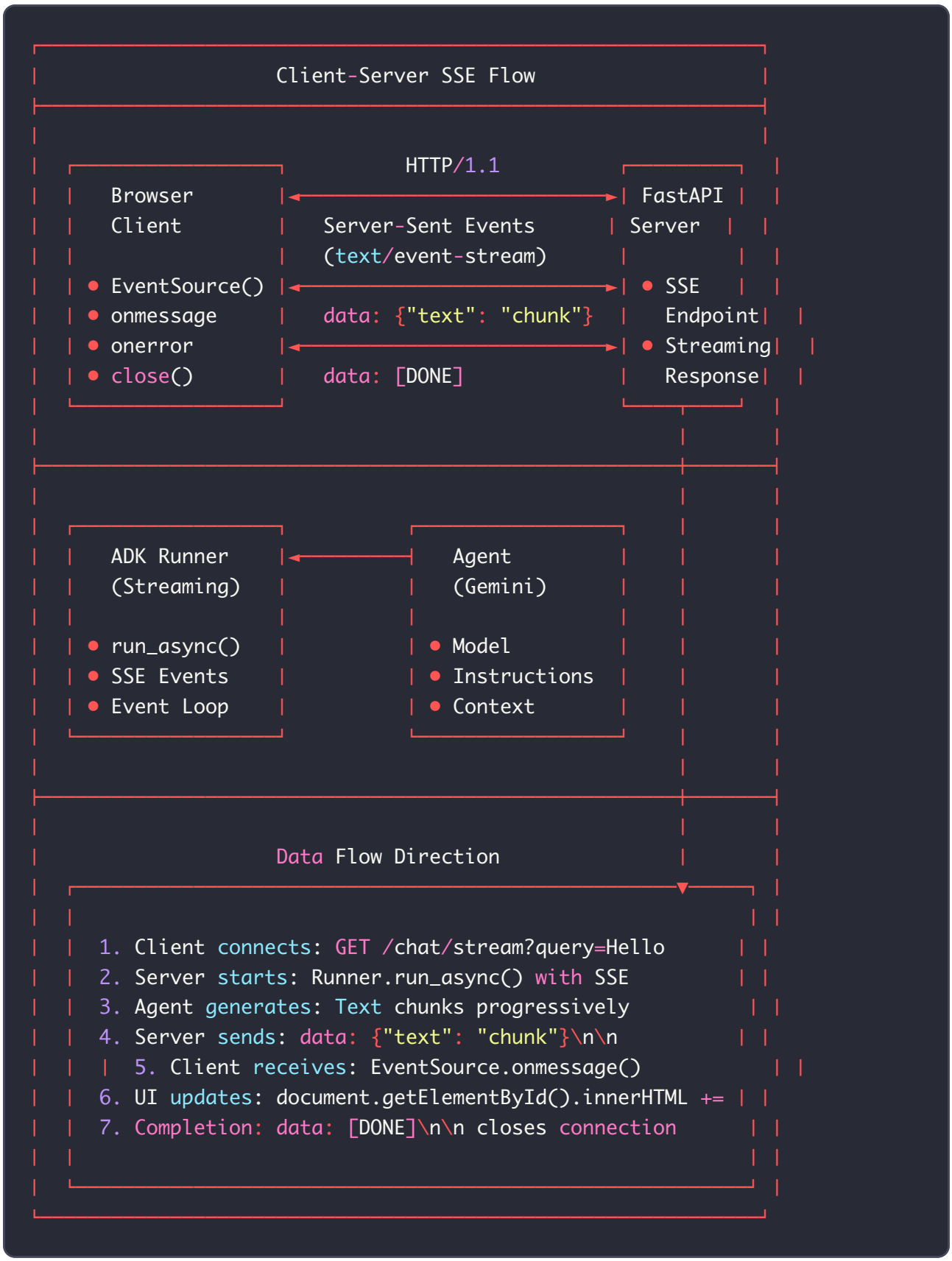
6. Building Web APIs with Streaming

FastAPI SSE Endpoint

```
"""
FastAPI endpoint with SSE streaming.
"""

from fastapi import FastAPI
from fastapi.responses import StreamingResponse
from google.adk.agents import Agent, Runner, RunConfig, StreamingMode
from google.adk.cli.adk_web_server import AdkWebServer
import json

app = FastAPI()

# Create agent
agent = Agent(
    model='gemini-2.0-flash',
    name='api_assistant'
)

runner = Runner()

async def generate_stream(query: str):
    """Generate SSE stream."""

    run_config = RunConfig(streaming_mode=StreamingMode.SSE)

    async for event in runner.run_async(query, agent=agent, run_config=run_con
        if event.content and event.content.parts:
            # Format as SSE
            chunk = event.content.parts[0].text
            data = json.dumps({'text': chunk})
            yield f"data: {data}\n\n"

    # Send completion signal
    yield "data: [DONE]\n\n"

@app.post("/chat/stream")
async def chat_stream(query: str):
    """Streaming chat endpoint."""

    return StreamingResponse(
        generate_stream(query),
        media_type="text/event-stream"
    )

# Usage with ADK web server
if __name__ == '__main__':
```

```
# adk api_server automatically sets up streaming endpoints
import uvicorn
uvicorn.run(app, host="0.0.0.0", port=8000)
```

Web API SSE Architecture:

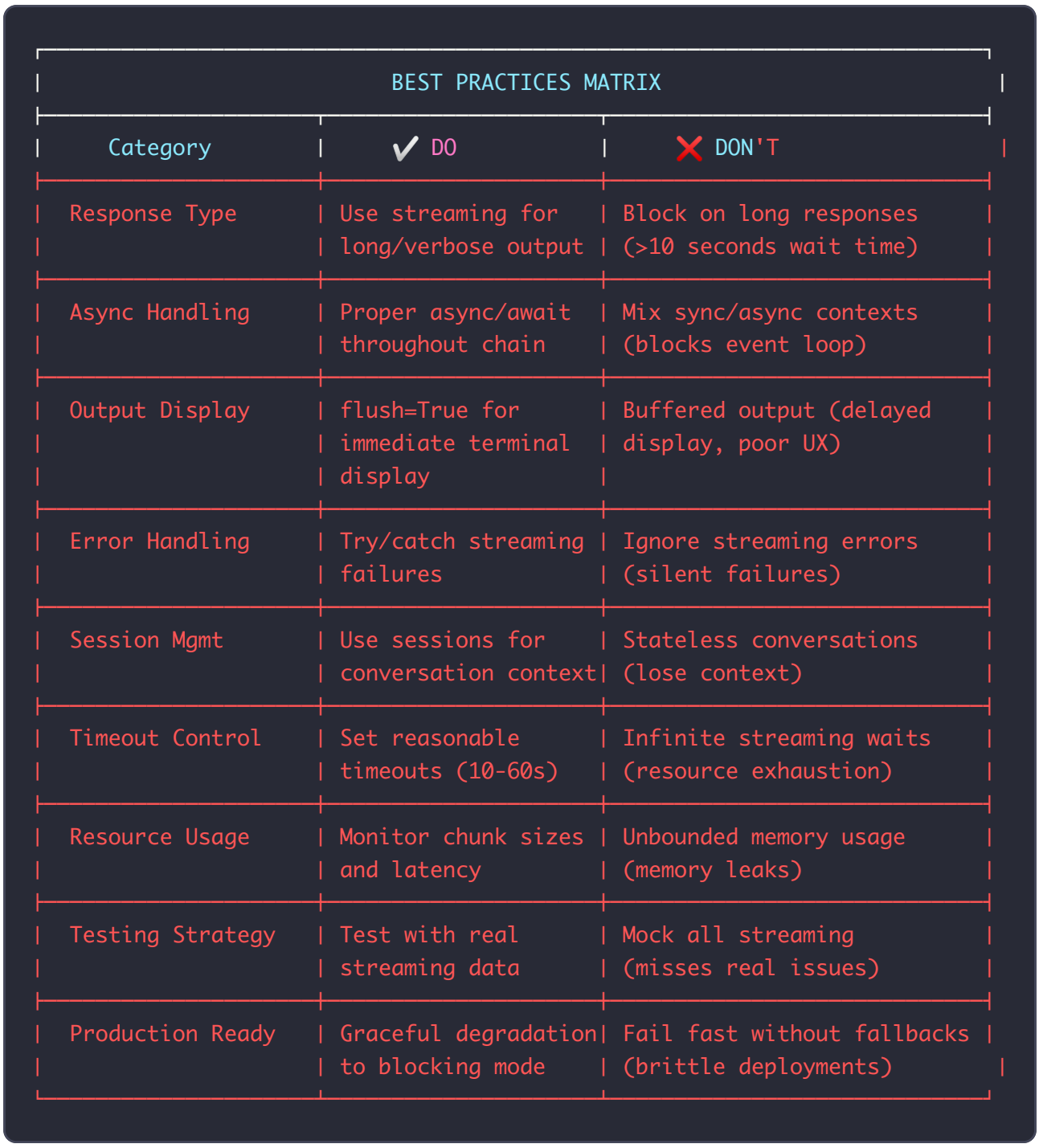

Client-Server SSE Flow
HTTP/1.1
Browser Client
Server-Sent Events (text/event-stream)
FastAPI Server
EventSource()
onmessage
onerror
close()
data: {"text": "chunk"}
data: [DONE]
SSE Endpoint
Streaming Response
ADK Runner (Streaming)
run_async()
SSE Events
Event Loop
Agent (Gemini)
Model
Instructions
Context
Data Flow Direction
1. Client connects: GET /chat/stream?query=Hello
2. Server starts: Runner.run_async() with SSE
3. Agent generates: Text chunks progressively
4. Server sends: data: {"text": "chunk"}\n\n
5. Client receives: EventSource.onmessage()
6. UI updates: document.getElementById().innerHTML +=
7. Completion: data: [DONE]\n\n closes connection

Client-Side JavaScript

```
// Connect to SSE endpoint
const eventSource = new EventSource(
  "http://localhost:8000/chat/stream?query=Hello"
);

eventSource.onmessage = (event) => {
  if (event.data === "[DONE]") {
    eventSource.close();
    return;
  }

  const data = JSON.parse(event.data);
  // Display chunk in UI
  document.getElementById("response").innerHTML += data.text;
};

eventSource.onerror = (error) => {
  console.error("SSE Error:", error);
  eventSource.close();
};
```

7. Best Practices

Streaming Implementation Guidelines:


BEST PRACTICES MATRIX
Category
✔ DO
❌ DON'T
Response Type
Use streaming for long/verbose output
Block on long responses (>10 seconds wait time)
Async Handling
Proper async/await throughout chain
Mix sync/async contexts (blocks event loop)
Output Display
flush=True for immediate terminal display
Buffered output (delayed display, poor UX)
Error Handling
Try/catch streaming failures
Ignore streaming errors (silent failures)
Session Mgmt
Use sessions for conversation context
Stateless conversations (lose context)
Timeout Control
Set reasonable timeouts (10-60s)
Infinite streaming waits (resource exhaustion)
Resource Usage
Monitor chunk sizes and latency
Unbounded memory usage (memory leaks)
Testing Strategy
Test with real streaming data
Mock all streaming (misses real issues)
Production Ready
Graceful degradation to blocking mode
Fail fast without fallbacks (brittle deployments)

✔️ DO: Use Streaming for Long Responses

```
# ✔️ Good - Stream long responses
run_config = RunConfig(streaming_mode=StreamingMode.SSE)

async for event in runner.run_async(
    "Write a detailed essay on climate change",
    agent=agent,
    run_config=run_config
):
    display_chunk(event)

# ❌ Bad - Blocking for long response
result = runner.run(
    "Write a detailed essay on climate change",
    agent=agent
)
# User waits 10+ seconds for complete response
```

✔️ DO: Handle Async Properly

```
# ✔️ Good - Proper async handling
async def handle_stream():
    async for event in runner.run_async(...):
        await process_event(event)

asyncio.run(handle_stream())

# ❌ Bad - Blocking in async context
async def handle_stream():
    result = runner.run(...)  # Blocks async loop
```

✔️ DO: Flush Output Immediately

```
# ✔️ Good - Flush for immediate display
print(chunk, end='', flush=True)

# ❌ Bad - Buffered output (delayed display)
print(chunk, end='')  # No flush
```

✔️ DO: Handle Streaming Errors

```
# ✔️ Good - Error handling
async def safe_stream(query, agent):
    try:
        run_config = RunConfig(streaming_mode=StreamingMode.SSE)

        async for event in runner.run_async(query, agent=agent, run_config=run
            if event.content and event.content.parts:
                print(event.content.parts[0].text, end='', flush=True)

    except Exception as e:
        print(f"\n[Error during streaming: {e}]")

# ❌ Bad - No error handling
async for event in runner.run_async(...):
    print(event.content.parts[0].text)  # Crashes on error
```

✔️ DO: Use Sessions for Context

```
# ✔️ Good - Session maintains conversation context
session = Session()

for message in conversation:
    async for event in runner.run_async(
        message,
        agent=agent,
        session=session,  # Context preserved
        run_config=run_config
    ):
        process_event(event)

# ❌ Bad - No session (loses context)
for message in conversation:
    async for event in runner.run_async(message, agent=agent, run_config=run_c
        process_event(event)
```

8. Troubleshooting

Issue: "Streaming classes not available"

Problem: Import errors for `StreamingMode`, `Runner`, `Session`, or `RunConfig`

Solution: Ensure you're using ADK v1.16.0 or later. These classes are available in current ADK:

```
# ✔️ Working with ADK v1.16.0+
from google.adk.agents import Agent
from google.adk.runners import Runner
from google.adk.agents.run_config import RunConfig, StreamingMode
from google.adk.sessions import InMemorySessionService
from google.genai import types

# Use the working implementation
from streaming_agent import stream_agent_response

async for chunk in stream_agent_response("Hello"):
    print(chunk, end='', flush=True)
```

Issue: "No streaming happening"

Problem: Response appears all at once instead of progressively

Solutions:

1. **Check ADK version**:

```
pip show google-adk  # Should be 1.16.0 or later
```

1. **Verify streaming configuration**:

```
# ✔️ Correct streaming config
run_config = RunConfig(streaming_mode=StreamingMode.SSE)

# ❌ Wrong - no streaming
run_config = RunConfig(streaming_mode=StreamingMode.NONE)
```

1. **Check output flushing**:

```
# ✔️ Flushed immediately
print(chunk, end='', flush=True)

# ❌ Buffered (delayed display)
print(chunk, end='')
```

Issue: "Agent.run_async method not found"

Problem: `AttributeError: 'LlmAgent' object has no attribute 'run_async'`

Solution: The Agent class uses `Runner.run_async()`. The working implementation uses real ADK streaming with fallback to simulation:

```
# ✔️ Working approach - uses real ADK streaming
from streaming_agent import stream_agent_response

async for chunk in stream_agent_response("Hello"):
    print(chunk, end='', flush=True)

# ❌ Won't work - Agent doesn't have run_async directly
# agent.run_async(query)  # AttributeError
```

Issue: "Streaming performance issues"

Problem: Real streaming feels too slow or has performance issues

Solution: The implementation uses real ADK streaming. If performance issues occur, the code falls back to simulated streaming. Check your network and API key:

```
# Check ADK version
pip show google-adk  # Should be 1.16.0+

# Test basic connectivity
python -c "import google.genai; print('GenAI import OK')"
```

If real streaming fails, you'll see a warning and fallback to simulation.

Issue: "Import errors with google.adk.agents"

Problem: Cannot import expected classes from ADK

Solution: Check your ADK version and use the working implementation:

```
# Check ADK version
pip show google.adk.agents

# Use working implementation instead
cd tutorial_implementation/tutorial14
python -c "from streaming_agent import stream_agent_response"
```

9. Testing Streaming Agents

Unit Tests

```python
import pytest
from google.adk.agents import Agent, Runner, RunConfig, StreamingMode

@pytest.mark.asyncio
async def test_streaming_response():
    """Test that streaming returns multiple chunks."""

    agent = Agent(
        model='gemini-2.0-flash',
        instruction='Provide detailed responses.'
    )

    runner = Runner()
    run_config = RunConfig(streaming_mode=StreamingMode.SSE)

    chunks = []

    async for event in runner.run_async(
        "Explain machine learning in detail",
        agent=agent,
        run_config=run_config
    ):
        if event.content and event.content.parts:
            chunks.append(event.content.parts[0].text)

    # Should receive multiple chunks
    assert len(chunks) > 1

    # Complete text should be reasonable length
    complete = ''.join(chunks)
    assert len(complete) > 100

@pytest.mark.asyncio
async def test_streaming_aggregation():
    """Test that streaming chunks combine correctly."""

    agent = Agent(model='gemini-2.0-flash')
    runner = Runner()
    run_config = RunConfig(streaming_mode=StreamingMode.SSE)

    chunks = []

    async for event in runner.run_async(
        "Count to 10",
        agent=agent,
        run_config=run_config
```

```
    ):
        if event.content and event.content.parts:
            chunks.append(event.content.parts[0].text)

    complete = ''.join(chunks)

    # Should contain all numbers
    for i in range(1, 11):
        assert str(i) in complete
```

Summary

You've mastered streaming responses with SSE:

Key Takeaways:

- ✔️ `StreamingMode.SSE` enables progressive response output
- ✔️ Use `RunConfig` to configure streaming
- ✔️ `runner.run_async()` with `async for` for streaming
- ✔️ Better UX - users see progress immediately
- ✔️ Essential for long responses and real-time applications
- ✔️ Works with sessions for conversation context
- ✔️ Can combine with tools and code execution
- ✔️ `flush=True` for immediate terminal output

Production Checklist:

- [ ] Using `RunConfig(streaming_mode=StreamingMode.SSE)`
- [ ] Proper async/await handling
- [ ] Error handling for streaming failures
- [ ] Session management for context
- [ ] Output flushing (`flush=True`)
- [ ] Timeout protection for long streams
- [ ] Testing streaming with multiple queries
- [ ] Monitoring chunk sizes and latency

Next Steps:

- **Tutorial 15**: Learn Live API for bidirectional streaming with audio
- **Tutorial 16**: Explore MCP Integration for extended tool ecosystem
- **Tutorial 17**: Implement Agent-to-Agent (A2A) communication

Resources:

- ADK Streaming Docs (https://google.github.io/adk-docs/streaming/)
- Server-Sent Events Standard (https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events)
- FastAPI SSE (https://fastapi.tiangolo.com/advanced/custom-response/#streamingresponse)

🎉 **Tutorial 14 Complete!** You now know how to implement streaming responses for better user experience. Continue to Tutorial 15 to learn about bidirectional streaming with the Live API.

Generated on 2025-10-21 09:02:29 from 14_streaming_sse.md

Source: Google ADK Training Hub