# Arquitectura de integración y mapeo de eventos en Google Agent Development Kit: Un análisis técnico desde el endpoint SSE hasta el renderizado de interfaces generativas en Flutter

La evolución de los ecosistemas de inteligencia artificial ha transitado desde simples interfaces de chat textual hacia sistemas agénticos complejos capaces de razonar, utilizar herramientas y, fundamentalmente, generar interfaces de usuario dinámicas que se adaptan en tiempo real a la intención del usuario. En el centro de esta revolución tecnológica se encuentra el Google Agent Development Kit (ADK), una infraestructura diseñada para orquestar el ciclo de vida de los agentes de software modernos.1 Este reporte técnico analiza con exhaustividad el flujo de datos desde el endpoint /run\_sse, desglosando la estructura de los eventos JSON devueltos y estableciendo un mapeo riguroso hacia entidades de dominio en Dart, la lógica de negocio mediante el patrón BLoC y la capa de presentación en Flutter a través de componentes de interfaz generativa (GenUI).

## Fundamentos del tiempo de ejecución del ADK y el bucle de eventos

El tiempo de ejecución del ADK no opera de manera lineal, sino que se sustenta en un bucle de eventos (Event Loop) que facilita la comunicación asíncrona entre el orquestador, denominado Runner, y la lógica de ejecución que comprende agentes, herramientas y callbacks.3 Este diseño permite que un agente pueda pausar su ejecución para solicitar información externa, requerir una acción del usuario o esperar la respuesta de un modelo de lenguaje de gran escala (LLM), sin bloquear el hilo principal de ejecución ni la conexión con el cliente.

El Runner actúa como la autoridad central que gestiona una consulta de usuario a través del método run\_async. Su responsabilidad abarca la carga de la sesión mediante el SessionService, la preparación del InvocationContext y la coordinación con servicios periféricos como el ArtifactService para la gestión de archivos y el MemoryService para la persistencia a largo plazo.3 Cuando el agente emite un evento, el Runner lo procesa, realiza las actualizaciones necesarias en el estado de la sesión y lo propaga hacia el cliente a través de una conexión de Server-Sent Events (SSE).

### La naturaleza del endpoint /run\_sse y el protocolo de streaming

A diferencia de los endpoints REST tradicionales que devuelven una respuesta única, el endpoint /run\_sse del ADK utiliza el protocolo de Server-Sent Events para mantener un canal de comunicación unidireccional abierto desde el servidor hacia el cliente.5 Esta elección arquitectónica es fundamental para la experiencia de usuario en aplicaciones de IA generativa por varias razones técnicas. En primer lugar, permite el streaming de tokens, donde el texto generado por el modelo aparece gradualmente en la interfaz, reduciendo la latencia percibida.5 En segundo lugar, facilita la notificación inmediata de llamadas a herramientas (function calls) y actualizaciones de estado antes de que el agente finalice su tarea completa.3

| **Característica** | **Endpoint /run** | **Endpoint /run\_sse** |
| --- | --- | --- |
| Método de entrega | Respuesta HTTP estándar (JSON Array) | Stream de Server-Sent Events (text/event-stream) |
| Latencia de respuesta | Bloqueante hasta la finalización total | Reactiva, entrega eventos según se generan |
| Soporte de streaming | No nativo para tokens individuales | Soporta streaming de tokens mediante partial=True |
| Gestión de sesiones | Sincrónica dentro de la petición | Asincrónica, permite reconexión mediante IDs |
| Uso recomendado | Procesamiento batch o depuración CLI | Interfaces interactivas y aplicaciones móviles/web |

El endpoint /run\_sse acepta un payload JSON que incluye el nombre de la aplicación (app\_name), el identificador de usuario (user\_id), el identificador de sesión (session\_id) y el mensaje nuevo (new\_message).5 Una vez iniciada la conexión, el servidor emite una serie de bloques de datos con el prefijo data:, cada uno conteniendo un objeto JSON que representa un Event del ADK.10

## Análisis detallado del mapa de eventos JSON

El objeto Event es la unidad atómica de información en el ecosistema ADK. Su estructura está diseñada para ser polimórfica, permitiendo transportar desde simples mensajes de texto hasta complejas definiciones de interfaces de usuario o solicitudes de credenciales de seguridad.3

### Estructura jerárquica del objeto de evento

Cada evento emitido por el endpoint /run\_sse contiene campos críticos para la trazabilidad y el procesamiento en el lado del cliente. Un evento típico presenta la siguiente configuración de campos:

* **id**: Un identificador único (UUID) para el evento, esencial para evitar el procesamiento duplicado en el cliente durante reconexiones.10
* **invocation\_id**: Un ID que vincula todos los eventos pertenecientes a un mismo "giro" de la conversación o ejecución de tarea.10
* **author**: Cadena de texto que identifica al emisor del evento, como user, agent, o nombres específicos de agentes secundarios en sistemas multi-agente.3
* **timestamp**: Marca de tiempo en formato Unix (segundos desde la época) que permite al cliente ordenar cronológicamente la historia de la sesión.10
* **partial**: Un booleano que, cuando es verdadero, indica que el contenido del evento es incompleto y forma parte de una ráfaga de streaming (como un fragmento de oración).3
* **content**: El objeto principal de carga útil, que sigue el esquema de los modelos Gemini de Google, compuesto por un rol y una lista de partes.9
* **actions**: Un contenedor de efectos secundarios que el agente desea aplicar, incluyendo cambios en el estado (state\_delta) o en los artefactos del sistema (artifact\_delta).3

### El polimorfismo de las partes de contenido

El campo content es donde reside la inteligencia del mensaje. Está diseñado para contener una lista de parts, cada una representando un tipo de dato diferente. Esta estructura permite que un solo evento contenga, por ejemplo, una explicación textual y simultáneamente una llamada a una herramienta de búsqueda.8

| **Tipo de Parte** | **Campo JSON Clave** | **Propósito y Relevancia** |
| --- | --- | --- |
| Texto | text | Contenido narrativo generado por el agente o el usuario. |
| Llamada a Función | function\_call | Solicitud del agente para ejecutar una herramienta específica con argumentos. |
| Respuesta de Función | function\_response | El resultado de una herramienta ejecutada que se devuelve al agente. |
| Solicitud de Auth | adk\_request\_credential | Evento especial para iniciar flujos de OAuth o autenticación.8 |
| A2UI Message | ---a2ui\_JSON--- | Delimitador convencional para incrustar protocolos de UI generativa en texto.12 |

### Mecanismos de actualización de estado: state\_delta y artifact\_delta

Una de las innovaciones más significativas del ADK es la capacidad de los agentes para modificar el estado de la aplicación de forma declarativa. El campo state\_delta contiene un diccionario de cambios que el Runner debe aplicar al SessionService.3 Estos cambios pueden incluir desde la actualización de la ubicación del usuario hasta la modificación de preferencias de presupuesto en un agente de viajes.13 El cliente debe ser capaz de procesar estos deltas para mantener una representación fiel de la "verdad" del agente en la interfaz de usuario.

El artifact\_delta funciona de manera similar pero orientado a objetos de datos más pesados o archivos persistentes, como documentos generados o imágenes procesadas.3 Estos deltas son fundamentales para sistemas que requieren seguimiento de fuentes y citaciones, aunque se ha reportado que en versiones actuales del ADK (como la 1.4.2), el procesamiento de metadatos de fundamentación (grounding) puede presentar inconsistencias en modo streaming comparado con el modo sincrónico.14

## El protocolo A2UI: Transformando JSON en interfaces

Para que el ADK pueda renderizar componentes visuales en Flutter, utiliza el protocolo Agent-to-User Interface (A2UI), una especificación diseñada para que los agentes describan interfaces de forma declarativa utilizando JSON en lugar de código imperativo.12 A2UI actúa como un contrato entre el backend del agente y el frontend de la aplicación.

### Ciclo de vida de una superficie generativa

El protocolo A2UI introduce el concepto de "superficies", que son áreas designadas en la pantalla donde el agente tiene control total sobre el contenido.15 El ciclo de vida de una interacción A2UI sigue una secuencia de mensajes específica transportada dentro de los eventos del ADK 12:

1. **createSurface**: El agente emite este mensaje para solicitar la creación de un nuevo contenedor de UI, asignándole un surfaceId único.16
2. **updateComponents**: Proporciona la estructura inicial de la interfaz, enviando una lista plana de componentes (botones, tarjetas, listas) y definiendo su jerarquía mediante IDs de referencia.16
3. **dataModelUpdate**: Permite al agente actualizar los valores de los datos vinculados a la interfaz (como el precio en una tarjeta de reserva) sin necesidad de reenviar toda la estructura de la UI. Utiliza punteros JSON para identificar el nodo exacto del modelo de datos que ha cambiado.16
4. **deleteSurface**: Indica que la tarea ha finalizado y la superficie de UI debe ser removida para limpiar el espacio visual del usuario.12

Este enfoque resuelve el problema de la "UI alucinada", ya que el agente no genera código Flutter directamente, sino que selecciona componentes de un catálogo predefinido y seguro que el desarrollador ha implementado previamente.18

## Mapeo a Entidades de Dominio en Dart

El primer paso para integrar el ADK en una aplicación Flutter es la creación de un modelo de dominio robusto que pueda deserializar el JSON del stream de forma segura. Dado el carácter polimórfico de los eventos, se recomienda una estructura de clases que utilice el patrón de diseño de visitantes o tipos de unión.

### Definición de la clase AdkEvent

A continuación se presenta una implementación técnica de la entidad de dominio AdkEvent en Dart, integrando los campos identificados en el análisis del mapa de eventos.

Dart

import 'dart:convert';  
  
/// Representa el evento base devuelto por el endpoint /run\_sse del ADK.  
class AdkEvent {  
 final String id;  
 final String? invocationId;  
 final String author;  
 final DateTime timestamp;  
 final bool partial;  
 final AdkContent content;  
 final Map<String, dynamic> actions;  
  
 AdkEvent({  
 required this.id,  
 this.invocationId,  
 required this.author,  
 required this.timestamp,  
 required this.content,  
 this.actions = const {},  
 this.partial = false,  
 });  
  
 factory AdkEvent.fromJson(Map<String, dynamic> json) {  
 return AdkEvent(  
 id: json['id'] as String,  
 invocationId: json['invocationId'] as String?,  
 author: json['author'] as String,  
 timestamp: DateTime.fromMillisecondsSinceEpoch(  
 ((json['timestamp'] as num) \* 1000).toInt(),  
 ),  
 partial: json['partial'] as bool??? false,  
 content: AdkContent.fromJson(json['content'] as Map<String, dynamic>),  
 actions: json['actions'] as Map<String, dynamic>??? {},  
 );  
 }  
}  
  
/// Contenedor polimórfico para el contenido del evento.  
class AdkContent {  
 final String? role;  
 final List<AdkPart> parts;  
  
 AdkContent({this.role, required this.parts});  
  
 factory AdkContent.fromJson(Map<String, dynamic> json) {  
 return AdkContent(  
 role: json['role'] as String?,  
 parts: (json['parts'] as List)  
 .map((p) => AdkPart.fromJson(p as Map<String, dynamic>))  
 .toList(),  
 );  
 }  
}

### Implementación de las Partes de Contenido

Para manejar los diferentes tipos de partes (texto, llamadas a funciones), se emplea una jerarquía de clases que facilita el procesamiento posterior en la capa de lógica de negocio.

Dart

abstract class AdkPart {  
 factory AdkPart.fromJson(Map<String, dynamic> json) {  
 if (json.containsKey('text')) {  
 return TextPart(text: json['text'] as String);  
 } else if (json.containsKey('functionCall')) {  
 return FunctionCallPart.fromJson(json['functionCall']);  
 } else if (json.containsKey('functionResponse')) {  
 return FunctionResponsePart.fromJson(json);  
 }  
 return UnknownPart();  
 }  
}  
  
class TextPart implements AdkPart {  
 final String text;  
 TextPart({required this.text});  
   
 /// Determina si el texto contiene una carga útil de A2UI.  
 bool get containsA2ui => text.contains('---a2ui\_JSON---');  
}  
  
class FunctionCallPart implements AdkPart {  
 final String name;  
 final Map<String, dynamic> args;  
 final String? id;  
  
 FunctionCallPart({required this.name, required this.args, this.id});  
  
 factory FunctionCallPart.fromJson(Map<String, dynamic> json) {  
 return FunctionCallPart(  
 name: json['name'] as String,  
 args: json['args'] as Map<String, dynamic>??? {},  
 id: json['id'] as String?,  
 );  
 }  
}

## Implementación de la Capa de Lógica de Negocio: El Patrón BLoC

El BLoC (Business Logic Component) es el encargado de orquestar el flujo desde el repositorio que consume el stream SSE hasta la actualización del estado de la interfaz. Su función principal es transformar eventos de entrada (como enviar un mensaje) en estados de salida que la UI de Flutter pueda consumir reactivamente.17

### Definición del Estado del GenUiBloc

El estado debe ser capaz de rastrear la historia de la conversación, el estado de carga y, crucialmente, las superficies de A2UI activas que deben ser renderizadas por widgets específicos.

Dart

import 'package:flutter\_bloc/flutter\_bloc.dart';  
  
class GenUiState {  
 final List<AdkEvent> history;  
 final bool isStreaming;  
 final Map<String, String> activeSurfaces; // ID de superficie -> JSON de componentes  
 final String? errorMessage;  
  
 GenUiState({  
 this.history = const,  
 this.isStreaming = false,  
 this.activeSurfaces = const {},  
 this.errorMessage,  
 });  
  
 GenUiState copyWith({  
 List<AdkEvent>? history,  
 bool? isStreaming,  
 Map<String, String>? activeSurfaces,  
 String? errorMessage,  
 }) {  
 return GenUiState(  
 history: history?? this.history,  
 isStreaming: isStreaming?? this.isStreaming,  
 activeSurfaces: activeSurfaces?? this.activeSurfaces,  
 errorMessage: errorMessage?? this.errorMessage,  
 );  
 }  
}

### Gestión del Stream SSE en el BLoC

La implementación del BLoC debe manejar la asincronía del stream SSE. Cuando se recibe un evento, el BLoC no solo actualiza la historia, sino que también inspecciona si el evento contiene instrucciones de A2UI para actualizar la interfaz dinámica.

Dart

class GenUiBloc extends Bloc<GenUiEvent, GenUiState> {  
 final AdkRepository repository;  
  
 GenUiBloc(this.repository) : super(GenUiState()) {  
 on<SendMessage>((event, emit) async {  
 emit(state.copyWith(isStreaming: true));  
   
 try {  
 final stream = repository.runSseStream(event.text);  
   
 await emit.forEach<AdkEvent>(  
 stream,  
 onData: (adkEvent) {  
 final updatedHistory = List<AdkEvent>.from(state.history)..add(adkEvent);  
 final updatedSurfaces = Map<String, String>.from(state.activeSurfaces);  
   
 // Procesamiento de mensajes A2UI incrustados en partes de texto  
 for (var part in adkEvent.content.parts) {  
 if (part is TextPart && part.containsA2ui) {  
 final a2uiJson = \_extractA2ui(part.text);  
 \_handleA2uiProtocol(a2uiJson, updatedSurfaces);  
 }  
 }  
   
 return state.copyWith(  
 history: updatedHistory,  
 activeSurfaces: updatedSurfaces,  
 );  
 },  
 );  
 } catch (e) {  
 emit(state.copyWith(errorMessage: e.toString()));  
 } finally {  
 emit(state.copyWith(isStreaming: false));  
 }  
 });  
 }  
  
 String \_extractA2ui(String text) {  
 return text.split('---a2ui\_JSON---').last.trim();  
 }  
  
 void \_handleA2uiProtocol(String jsonStr, Map<String, String> surfaces) {  
 final Map<String, dynamic> data = jsonDecode(jsonStr);  
   
 // El protocolo A2UI define mensajes como createSurface o updateComponents   
 if (data.containsKey('createSurface')) {  
 final surfaceId = data['surfaceId'];  
 surfaces[surfaceId] = jsonStr;  
 } else if (data.containsKey('deleteSurface')) {  
 surfaces.remove(data['surfaceId']);  
 }  
 }  
}

## Capa de Presentación: Widgets de Interfaz Generativa

En Flutter, la renderización de los eventos del ADK se divide en dos categorías: el flujo de chat convencional y las superficies de interfaz generativa (GenUI). El SDK de Google proporciona el widget GenUiSurface para este propósito, el cual se vincula a un GenUiManager y un catálogo de componentes.17

### Configuración del Catálogo de Componentes

Antes de renderizar widgets dinámicos, el desarrollador debe definir qué componentes puede "invocar" el agente. Cada componente se registra como un CatalogItem, vinculando un esquema de datos JSON con un constructor de widgets en Flutter.18

Dart

import 'package:genui/genui.dart';  
import 'package:flutter/material.dart';  
  
final myWidgetCatalog =,  
 ),  
 widgetBuilder: ({required data, required id,...}) {  
 final json = data as Map<String, dynamic>;  
 return Card(  
 color: Colors.amber.shade50,  
 child: Padding(  
 padding: EdgeInsets.all(16),  
 child: Column(  
 crossAxisAlignment: CrossAxisAlignment.start,  
 children:, style: TextStyle(fontSize: 18, fontWeight: FontWeight.bold)),  
 SizedBox(height: 10),  
 Text(json['answer'], style: TextStyle(fontStyle: FontStyle.italic)),  
 ],  
 ),  
 ),  
 );  
 },  
 ),  
];

### El Widget de Superficie Generativa

En la pantalla principal, el BLoC provee la lista de eventos. Si un evento contiene una referencia a una superficie de UI, se utiliza GenUiSurface para inyectar el componente dinámico en el árbol de widgets.18

Dart

class AdkChatView extends StatelessWidget {  
 @override  
 Widget build(BuildContext context) {  
 return BlocBuilder<GenUiBloc, GenUiState>(  
 builder: (context, state) {  
 return Column(  
 children:;  
   
 // Renderizado condicional basado en el tipo de contenido  
 return Column(  
 children:,  
 );  
 },  
 ),  
 ),  
 if (state.isStreaming) LinearProgressIndicator(),  
 \_MessageInput(),  
 ],  
 );  
 },  
 );  
 }  
}

## Manejo de Interacciones: Del Widget de Regreso al Agente

La arquitectura GenUI no es una vía de sentido único. Cuando un usuario interactúa con un widget generado (por ejemplo, al hacer clic en un botón de "Aprobar" en un formulario dinámico), esta acción debe comunicarse de regreso al agente para que este pueda razonar y responder.15

### El Flujo de UserActionEvent

Cuando el widget dispara un evento, este viaja a través del pipeline de GenUI siguiendo estos pasos técnicos 17:

1. **Captura**: El widget dentro del catálogo llama a la función dispatchEvent.
2. **Inyección de Contexto**: El widget GenUiSurface intercepta el evento e inyecta automáticamente el surfaceId para que el agente sepa de dónde proviene la interacción.
3. **Encapsulación**: El A2uiMessageProcessor envuelve el evento en un sobre de protocolo userAction que incluye el nombre de la acción, el ID del componente de origen y un timestamp.21
4. **Transmisión**: El ContentGenerator (ya sea vía FirebaseAi o A2uiContentGenerator) envía este JSON estructurado de regreso al agente a través de una nueva llamada al endpoint de ejecución.20

| **Campo de userAction** | **Tipo** | **Descripción** |
| --- | --- | --- |
| name | String | Identificador de la acción definido por el agente (ej. submit\_form). |
| surfaceId | String | El contenedor donde ocurrió la acción. |
| sourceComponentId | String | El ID del componente específico que disparó el evento. |
| context | JSON | Datos adicionales del estado del widget al momento del evento. |

Este bucle cerrado permite que el agente trate las interacciones del usuario como señales estructuradas, no simplemente como texto plano, lo que facilita una lógica de control de flujo mucho más precisa y menos propensa a errores de interpretación.15

## Desafíos Técnicos y Consideraciones de Implementación

La integración del ADK y GenUI presenta desafíos que deben ser abordados para lograr una aplicación de nivel empresarial.

### Autenticación y Seguridad (ADK Request Credential)

Si el agente intenta ejecutar una herramienta que requiere permisos (como acceder al calendario del usuario), emitirá una function\_call especial llamada adk\_request\_credential.8 El mapeo de eventos debe estar preparado para detectar este ID de función y extraer la AuthConfig. El cliente Flutter debe entonces pausar el flujo del stream, abrir una vista web o una pestaña del sistema para la autenticación OAuth, capturar el código de redirección y reanudar la interacción del agente enviando la credencial obtenida.8

### Problemas de Codificación en el Transporte

Se han identificado escenarios, particularmente en implementaciones con Agent Engine de Google Cloud, donde las cadenas de SSE pueden sufrir una doble codificación JSON.22 Esto ocurre cuando el servidor serializa una cadena que ya es un JSON SSE, resultando en datos como data: "data: {...}\n\n". El cliente debe implementar una lógica defensiva en su interceptor de red para detectar y normalizar estas estructuras de "SSE dentro de JSON" antes de pasarlas a la entidad de dominio.22

### Observabilidad y Depuración

Dado que el comportamiento del agente depende del historial y del estado dinámico, es vital implementar un sistema de logging exhaustivo. El SDK de Flutter permite configurar niveles de log globales para capturar cada mensaje que fluye a través del GenUiManager.23 Herramientas de observabilidad externas como Comet Opik pueden integrarse en el lado del servidor para capturar las trazas de razonamiento que no siempre se envían al cliente pero que explican por qué el agente generó un componente de UI específico.5

## Conclusión sobre la Convergencia Tecnológica de ADK y Flutter

El ecosistema del Google Agent Development Kit, en conjunción con el SDK de GenUI para Flutter, representa un cambio de paradigma en el desarrollo de software. Al mapear sistemáticamente los eventos JSON del endpoint /run\_sse hacia entidades de dominio Dart y estados de BLoC, los desarrolladores pueden construir aplicaciones donde la interfaz de usuario no es una rejilla estática, sino un lienzo vivo y adaptativo.

La arquitectura analizada demuestra que la clave del éxito en las interfaces generativas reside en la separación de responsabilidades: el agente decide el "qué" y el "cuándo" de la interacción a través de mensajes declarativos, mientras que la aplicación Flutter mantiene el control sobre el "cómo" mediante un catálogo de componentes seguros y performantes. Esta sinergia no solo reduce la latencia y mejora la satisfacción del usuario, sino que establece una base sólida para el futuro de las aplicaciones inteligentes, donde la frontera entre la conversación y la acción se vuelve invisible para el usuario final.2

#### Works cited

1. Building agents with the ADK and the new Interactions API - Google Developers Blog, accessed on January 1, 2026, <https://developers.googleblog.com/building-agents-with-the-adk-and-the-new-interactions-api/>
2. Generative UI for Flutter: Build Adaptive, Branded, and Intelligent User Experiences, accessed on January 1, 2026, <https://www.verygood.ventures/blog/redefining-the-digital-booking-experience-with-generative-ui>
3. Agent Runtime - Agent Development Kit - Google, accessed on January 1, 2026, <https://google.github.io/adk-docs/runtime/>
4. Research Assistant Agent - Kaggle, accessed on January 1, 2026, <https://www.kaggle.com/code/emanafi/research-assistant-agent>
5. API Server - Agent Development Kit - Google, accessed on January 1, 2026, <https://google.github.io/adk-docs/runtime/api-server/>
6. Streaming server-sent events | Apigee - Google Cloud Documentation, accessed on January 1, 2026, <https://docs.cloud.google.com/apigee/docs/api-platform/develop/server-sent-events>
7. Rich and dynamic user interfaces with Flutter and generative UI | by Abdallah W Shaban, accessed on January 1, 2026, <https://blog.flutter.dev/rich-and-dynamic-user-interfaces-with-flutter-and-generative-ui-178405af2455>
8. Authentication - Agent Development Kit - Google, accessed on January 1, 2026, <https://google.github.io/adk-docs/tools-custom/authentication/>
9. REST API - Agent Development Kit - Google, accessed on January 1, 2026, <https://google.github.io/adk-docs/api-reference/rest/>
10. [ Question ] Google ADK Tools + VLLM Model + LiteLLM integration problem #1968 - GitHub, accessed on January 1, 2026, <https://github.com/google/adk-python/issues/1968>
11. Google Gen AI SDK documentation, accessed on January 1, 2026, <https://googleapis.github.io/python-genai/>
12. Build with Google's new A2UI Spec: Agent User Interfaces with A2UI + AG-UI - CopilotKit, accessed on January 1, 2026, <https://www.copilotkit.ai/blog/build-with-googles-new-a2ui-spec-agent-user-interfaces-with-a2ui-ag-ui>
13. Cicerone: an ADK Travel Agent with Google Maps grounding - Medium, accessed on January 1, 2026, <https://medium.com/google-cloud/cicerone-an-adk-travel-agent-with-google-maps-grounding-f5ac1fc5b483>
14. ADK Streaming Grounding Callback Bug Report · Issue #2230 · google/adk-python - GitHub, accessed on January 1, 2026, <https://github.com/google/adk-python/issues/2230>
15. A2UI: Understanding Agent-Driven Interfaces | by #TheGenAIGirl — code, community, and GenAI. | Google Cloud - Medium, accessed on January 1, 2026, <https://medium.com/google-cloud/a2ui-understanding-agent-driven-interfaces-2ce79201d27a>
16. A2UI Specification, accessed on January 1, 2026, <https://a2ui.org/specification/v0.9-a2ui/>
17. GenUI SDK main components and concepts - Flutter documentation, accessed on January 1, 2026, <https://docs.flutter.dev/ai/genui/components>
18. Get started with the GenUI SDK for Flutter, accessed on January 1, 2026, <https://docs.flutter.dev/ai/genui/get-started>
19. Google releases A2UI - How the new spec fits within the generative UI space : r/AI\_Agents, accessed on January 1, 2026, <https://www.reddit.com/r/AI_Agents/comments/1pocrep/google_releases_a2ui_how_the_new_spec_fits_within/>
20. genui | Flutter package - Pub.dev, accessed on January 1, 2026, <https://pub.dev/packages/genui>
21. Input and events - Flutter documentation, accessed on January 1, 2026, <https://docs.flutter.dev/ai/genui/input-events>
22. Feature Request: Agent Engine :streamQuery?alt=sse + AG-UI: yielding SSE strings gets JSON-encoded (double-encoding), breaking direct CopilotKit compatibility #2871 - GitHub, accessed on January 1, 2026, <https://github.com/CopilotKit/CopilotKit/issues/2871>
23. How to Use GenUI in Flutter to Build Dynamic, AI-Driven Interfaces - freeCodeCamp, accessed on January 1, 2026, <https://www.freecodecamp.org/news/how-to-use-genui-in-flutter-to-build-dynamic-ai-driven-interfaces/>