Comprehensive Research Document: Google Agent Development Kit (ADK)

Table of Contents

Introduction

The Google Agent Development Kit (ADK) is an open-source framework designed to simplify the development of AI agents. It provides a structured approach to building, testing, and deploying agents that can perform complex tasks through natural language interactions. The ADK addresses the challenges of agent development by offering a comprehensive set of tools, libraries, and best practices.

Core Purpose and Benefits

The primary purpose of the ADK is to provide developers with a unified framework for building AI agents that can:

1. **Understand and respond to natural language**: Process user queries and generate coherent, contextually appropriate responses.
2. **Use tools and APIs**: Interact with external systems and services to accomplish tasks.
3. **Maintain context across conversations**: Remember previous interactions and use that context to inform future responses.
4. **Execute complex workflows**: Perform multi-step processes that may involve multiple tools and decision points.

Key benefits of using the ADK include:

- **Reduced development complexity**: The ADK abstracts away many of the technical challenges of agent development, allowing developers to focus on defining agent behavior.
- **Standardized architecture**: The ADK provides a consistent structure for agent applications, making them easier to maintain and extend.
- **Improved reliability**: Built-in testing and evaluation tools help ensure agents perform as expected.
- **Scalability**: The ADK supports deployment options that can scale from prototype to production.

Target Users and Use Cases

The ADK is designed for:

1. **AI/ML Engineers**: Who want to build sophisticated agents without reinventing the wheel.
2. **Software Developers**: Who need to integrate AI capabilities into their applications.
3. **Researchers**: Who are exploring new agent architectures and capabilities.
4. **Enterprises**: Who want to deploy AI agents for internal or customer-facing applications.

Common use cases include:

- **Customer service agents**: Handling inquiries, troubleshooting issues, and providing information.
- **Personal assistants**: Managing schedules, sending messages, and performing tasks.
- **Knowledge workers**: Researching topics, summarizing information, and generating content.
- **Domain-specific experts**: Providing specialized knowledge and guidance in fields like healthcare, finance, or legal.

Getting Started with ADK

The ADK is available as a Python package that can be installed via pip:

```
pip install google-adk
```

A typical workflow for developing an agent with the ADK involves:

1. **Define the agent's purpose and capabilities**: Determine what tasks the agent should perform and what tools it will need.
2. **Set up the agent structure**: Create the agent class, define its properties, and configure its behavior.
3. **Add tools**: Integrate the tools the agent will use to accomplish tasks.
4. **Test and evaluate**: Use the ADK's testing and evaluation tools to ensure the agent performs as expected.
5. **Deploy**: Deploy the agent to a production environment.

The ADK provides a variety of examples and templates to help developers get started quickly.

Agent Types and Models

The ADK supports multiple agent types, each designed for different use cases and requirements. Understanding these agent types and their capabilities is essential for choosing the right approach for your specific needs.

LLM Agents: Architecture, Capabilities, Use Cases

LLM (Large Language Model) agents are the most common type of agent in the ADK. They leverage the power of large language models to understand and generate natural language, making them ideal for conversational interfaces.

Architecture

LLM agents in the ADK follow a structured architecture:

1. **Input Processing**: The agent receives user input and processes it to understand the intent.
2. **Context Management**: The agent maintains context across turns using the session and state mechanisms.
3. **Tool Selection**: Based on the user's intent, the agent selects appropriate tools to use.
4. **Tool Execution**: The agent executes the selected tools and processes their results.
5. **Response Generation**: The agent generates a natural language response based on the tool results and context.

The `LlmAgent` class in the ADK encapsulates this architecture, providing a framework for building agents that leverage language models.

Capabilities

LLM agents can:

- **Understand natural language**: Process and interpret user queries in natural language.
- **Use tools**: Execute tools to perform tasks and retrieve information.
- **Maintain context**: Remember previous interactions and use that context to inform future responses.
- **Generate coherent responses**: Produce natural language responses that are contextually appropriate.
- **Follow instructions**: Adhere to specific guidelines and constraints defined by the developer.

Use Cases

LLM agents are well-suited for:

- **Conversational interfaces**: Chatbots, virtual assistants, and other applications where natural language interaction is primary.
- **Complex task execution**: Workflows that involve multiple steps and decision points.
- **Information retrieval and synthesis**: Applications that need to gather, process, and present information from various sources.

Workflow Agents: Structure and Interactions

Workflow agents are designed to execute predefined sequences of steps, making them ideal for structured tasks with clear processes.

Structure

Workflow agents in the ADK are structured around:

1. **Workflow Definition**: A clear specification of the steps to be executed.
2. **State Management**: Tracking the current state of the workflow.
3. **Transition Logic**: Rules for moving from one step to another.
4. **Error Handling**: Mechanisms for dealing with exceptions and failures.

Interactions

Workflow agents interact with:

- **Users**: To receive input and provide updates on workflow progress.

- **Tools**: To execute specific actions required by the workflow.
- **Other Agents**: To delegate subtasks or collaborate on complex workflows.
- **External Systems**: To integrate with existing processes and data sources.

Custom Agents: Flexibility and Customization

The ADK allows developers to create custom agent types tailored to specific requirements.

Flexibility

Custom agents offer:

- **Architectural Freedom**: Developers can define their own agent architecture.
- **Specialized Behavior**: Agents can be optimized for specific domains or tasks.
- **Integration Capabilities**: Custom agents can be designed to integrate with existing systems and workflows.

Customization

Key areas for customization include:

- **Decision Logic**: How the agent decides what actions to take.
- **Context Management**: How the agent maintains and uses context.
- **Tool Integration**: How the agent interacts with tools and external systems.
- **Response Generation**: How the agent formulates responses to users.

Multi-Agent Systems: Development and Coordination

The ADK supports the development of multi-agent systems, where multiple agents collaborate to accomplish complex tasks.

Development

Developing multi-agent systems involves:

1. **Agent Specialization**: Defining the role and capabilities of each agent.
2. **Communication Protocols**: Establishing how agents will communicate with each other.
3. **Coordination Mechanisms**: Determining how agents will coordinate their actions.
4. **Conflict Resolution**: Handling situations where agents may have conflicting goals or actions.

Coordination

Coordination in multi-agent systems can be achieved through:

- **Hierarchical Structures**: Parent-child relationships between agents.
- **Peer-to-Peer Interactions**: Direct communication between agents at the same level.
- **Mediator Agents**: Specialized agents that coordinate the activities of other agents.
- **Shared State**: Common knowledge that all agents can access and update.

Compatible Language Models and Integration Methods

The ADK is designed to work with a variety of language models, providing flexibility in choosing the right model for your specific needs.

Compatible Models

The ADK supports:

- **Google Models**: Including Gemini and PaLM.
- **OpenAI Models**: Including GPT-3.5 and GPT-4.
- **Open-Source Models**: Various open-source language models can be integrated.
- **Custom Models**: Developers can integrate their own language models.

Integration Methods

Models can be integrated through:

1. **Direct API Integration**: Using the model's API directly.
2. **Model Adapters**: Using adapters provided by the ADK to standardize model interactions.
3. **Custom Implementations**: Creating custom integrations for specific models.

The ADK provides a consistent interface for working with different models, allowing developers to switch models without significant code changes.

Tools and Integrations

Tools are a fundamental component of the ADK, enabling agents to perform actions and access external systems. The ADK provides a rich ecosystem of tools and integration options to extend agent capabilities.

Function Tools: Integration of Custom Functions

Function tools allow developers to integrate custom Python functions into their agents, enabling them to perform specific tasks or access external systems.

Key Features

- **Simple Integration**: Easy to add custom functions to agents.
- **Type Safety**: Function parameters and return values can be typed for better reliability.
- **Error Handling**: Built-in mechanisms for handling function errors.
- **State Access**: Functions can access and modify session state.

Implementation

To create a function tool, developers define a Python function and wrap it with the `FunctionTool` class:

```python
from google.adk.tools import FunctionTool

def calculate_sum(a: int, b: int) -> int:
    """Calculate the sum of two numbers."""
    return a + b

sum_tool = FunctionTool(
    name="calculate_sum",
    description="Calculate the sum of two numbers",
    function=calculate_sum
)
```

Function tools can then be added to an agent's tool set, allowing the agent to use them when needed.

Built-in Tools: Pre-packaged Tools

The ADK includes a variety of pre-packaged tools that provide common functionality, saving developers the effort of implementing these capabilities themselves.

Available Built-in Tools

- **Web Search**: Search the web for information.
- **Calculator**: Perform mathematical calculations.
- **Date/Time**: Access and manipulate date and time information.
- **File Operations**: Read, write, and manage files.
- **System Information**: Access information about the system environment.

Usage

Built-in tools can be imported and added to an agent's tool set:

```
from google.adk.tools.builtin import WebSearchTool, CalculatorTool

agent = LlmAgent(
    name="my_agent",
    model=model,
    tools=[WebSearchTool(), CalculatorTool()]
)
```

Third-party Tools: Integration Process and Examples

The ADK supports integration with third-party tools and services, expanding the range of capabilities available to agents.

Integration Process

1. **API Access**: Obtain API credentials for the third-party service.
2. **Tool Implementation**: Create a tool that interfaces with the API.
3. **Authentication Configuration**: Set up authentication for the tool.
4. **Error Handling**: Implement error handling for API failures.

Examples

- **Weather Services**: Access weather forecasts and conditions.
- **Translation Services**: Translate text between languages.
- **E-commerce Platforms**: Search for products, check prices, and place orders.
- **Social Media APIs**: Post updates, retrieve information, and interact with social platforms.

Google Cloud Tools: Specific Services Integration

The ADK provides specialized tools for integrating with Google Cloud services, enabling agents to leverage the full power of Google's cloud platform.

Available Google Cloud Tools

- **BigQuery**: Query and analyze large datasets.
- **Cloud Storage**: Store and retrieve files.
- **Vertex AI**: Access machine learning models and services.
- **Google Calendar**: Manage calendar events and appointments.
- **Google Drive**: Access and manipulate files in Google Drive.

Implementation

Google Cloud tools typically require authentication and specific configuration:

```
from google.adk.tools.google_api_tool import calendar_tool_set

# Configure authentication
client_id = "YOUR_CLIENT_ID"
client_secret = "YOUR_CLIENT_SECRET"

# Get calendar tools
calendar_tools = calendar_tool_set.get_tools()

# Configure authentication for each tool
for tool in calendar_tools:
    tool.configure_auth(client_id=client_id, client_secret=client_secret)

# Add tools to agent
agent = LlmAgent(
    name="calendar_agent",
    model=model,
    tools=calendar_tools
)
```

MCP Tools: Purpose and Usage

MCP (Multi-Cloud Platform) tools in the ADK provide a unified interface for accessing services across different cloud platforms, simplifying multi-cloud deployments.

Purpose

- **Cloud Agnosticism**: Develop agents that can work across different cloud environments.
- **Simplified Integration**: Use a consistent interface regardless of the underlying cloud provider.
- **Portability**: Move agents between cloud platforms without significant code changes.

Usage

MCP tools are used similarly to other tools in the ADK, but with configuration specific to the target cloud platform:

```
from google.adk.tools.mcp import StorageTool

# Configure for specific cloud provider
```

```
storage_tool = StorageTool(provider="aws", config={...})

# Add to agent
agent = LlmAgent(
    name="storage_agent",
    model=model,
    tools=[storage_tool]
)
```

OpenAPI Tools: Integration with OpenAPI Specification

The ADK supports generating tools from OpenAPI specifications, enabling agents to interact with any API that provides an OpenAPI definition.

Key Features

- **Automatic Tool Generation**: Create tools directly from OpenAPI specs.
- **Comprehensive API Access**: Access all endpoints defined in the spec.
- **Type Safety**: Parameter and response types are derived from the spec.
- **Documentation Integration**: Tool descriptions are generated from API documentation.

Implementation

To create tools from an OpenAPI specification:

```
from google.adk.tools.openapi_tool import OpenAPIToolset

# Create toolset from OpenAPI spec
toolset = OpenAPIToolset(
    name="weather_api",
    description="Tools for accessing weather information",
    spec_path="/path/to/openapi.json"
)

# Get all tools from the toolset
weather_tools = toolset.get_tools()

# Add tools to agent
agent = LlmAgent(
    name="weather_agent",
    model=model,
    tools=weather_tools
)
```

This approach allows agents to interact with a wide range of APIs with minimal development effort.

Deployment and Authentication

Once you've built and tested your agent using ADK, the next step is to deploy it so it can be accessed, queried, and used in production or integrated with other applications. Deployment moves your agent from your local development machine to a scalable and reliable environment.

Deployment Options

Your ADK agent can be deployed to a range of different environments based on your needs for production readiness or custom flexibility:

Agent Engine in Vertex AI

Agent Engine is a fully managed Google Cloud service enabling developers to deploy, manage, and scale AI agents in production. Agent Engine handles the infrastructure to scale agents in production so you can focus on creating intelligent and impactful applications.

Key features: - Fully managed service on Google Cloud - Auto-scaling capabilities - Simplified deployment process - Integrated with Google Cloud ecosystem

Deployment process: 1. Install the Vertex AI SDK: `python pip install google-cloud-aiplatform[adk,agent_engines]`

1. Initialize and create your agent for deployment: ```python from vertexai import agent_engines

remote_app = agent_engines.create(agent_engine=root_agent, requirements=["google-cloud-aiplatform[adk,agent_engines]",]) ```

1. Test your agent locally before deployment
2. Deploy your agent to Agent Engine
3. Grant the deployed agent necessary permissions
4. Interact with your agent through the deployed endpoints

Requirements: - Agent Engine only supports Python version >=3.9 and <=3.12 - Proper authentication setup for Google Cloud

Cloud Run

Cloud Run is a fully managed platform that enables you to run your code directly on top of Google's scalable infrastructure.

Key features: - Containerized deployment - Scalable infrastructure - Pay-per-use pricing model - Support for various authentication methods

Deployment process: You can deploy your agent to Cloud Run using either:

1. The ADK CLI (recommended): `bash adk deploy cloud_run`
2. The gcloud CLI: `bash gcloud run deploy`

Authentication options during deployment: During the deployment process, you might be prompted:

```
Allow unauthenticated invocations to [your-service-name] (y/N)?
```

- Enter `y` to allow public access to your agent's API endpoint without authentication
- Enter `N` (or press Enter for the default) to require authentication (e.g., using an identity token)

Authentication Mechanisms

Many tools in ADK need to access protected resources (like user data in Google Calendar, Salesforce records, etc.) and require authentication. ADK provides a comprehensive system to handle various authentication methods securely.

Core Authentication Concepts

The key components involved in ADK authentication are:

1. **`AuthScheme`** : Defines how an API expects authentication credentials (e.g., as an API Key in a header, an OAuth 2.0 Bearer token). ADK supports the same types of authentication schemes as OpenAPI 3.0, using specific classes like:
2. `APIKey`
3. `HTTPBearer`
4. `OAuth2`
5. `OpenIdConnectWithConfig`
6. **`AuthCredential`** : Holds the initial information needed to start the authentication process (e.g., your application's OAuth Client ID/Secret, an API key value). It includes an `auth_type` specifying the credential type.

The general flow involves providing these details when configuring a tool. ADK then attempts to automatically exchange the initial credential for a usable one (like an access token) before the tool makes an API call. For flows requiring user interaction (like OAuth

consent), a specific interactive process involving the Agent Client application is triggered.

Supported Initial Credential Types

1. **API_KEY**:
2. For simple key/value authentication
3. Usually requires no exchange
4. Example: API keys for services like Google Maps or weather APIs
5. **HTTP**:
6. Can represent Basic Auth (not recommended/supported for exchange)
7. Can represent already obtained Bearer tokens
8. If it's a Bearer token, no exchange is needed
9. **OAUTH2**:
10. For standard OAuth 2.0 flows
11. Requires configuration (client ID, secret, scopes)
12. Often triggers the interactive flow for user consent
13. Common for Google APIs, social media platforms, etc.
14. **OPEN_ID_CONNECT**:
15. For authentication based on OpenID Connect
16. Similar to OAuth2, often requires configuration and user interaction
17. Provides identity verification in addition to authorization
18. **SERVICE_ACCOUNT**:
19. For Google Cloud Service Account credentials
20. Can use JSON key or Application Default Credentials
21. Typically exchanged for a Bearer token
22. Used for server-to-server authentication without user interaction

Configuring Authentication on Tools

Authentication is set up when defining your tool, with different approaches depending on the tool type:

1. **RestApiTool / OpenAPIToolset**:

2. Pass `auth_scheme` and `auth_credential` during initialization ```python from google.adk.tools.openapi_tool.auth.auth_helpers import token_to_scheme_credential

auth_scheme, auth_credential = token_to_scheme_credential("apikey", "query", "apikey", YOUR_API_KEY_STRING)

toolset = OpenAPIToolset(spec_path="path/to/spec.json", auth_scheme=auth_scheme, auth_credential=auth_credential) ```

1. **Google API Toolsets**:
2. Use the toolset's specific configuration method ```python from google.adk.tools.google_api_tool import calendar_tool_set

client_id = "YOUR_GOOGLE_OAUTH_CLIENT_ID.apps.googleusercontent.com" client_secret = "YOUR_GOOGLE_OAUTH_CLIENT_SECRET"

calendar_tools = calendar_tool_set.get_tools() for tool in calendar_tools: tool.configure_auth(client_id=client_id, client_secret=client_secret) ```

1. **APIHubToolset / ApplicationIntegrationToolset**:
2. Pass `auth_scheme` and `auth_credential` during initialization ```python from google.adk.tools.apihub_tool.apihub_toolset import APIHubToolset

auth_scheme, auth_credential = token_to_scheme_credential("apikey", "query", "apikey", YOUR_API_KEY_STRING)

api_toolset = APIHubToolset(name="sample-api", description="A tool using an API protected by API Key", apihub_resource_name="...", auth_scheme=auth_scheme, auth_credential=auth_credential) ```

Handling Interactive Authentication Flows

For authentication methods requiring user interaction (like OAuth consent screens), ADK implements a special flow:

1. The agent execution pauses when authentication is needed
2. A special event with the function call `adk_request_credential` is generated
3. The client application (your UI, CLI, etc.) must detect this event and handle the user interaction
4. After user completes authentication, the client provides the obtained credentials back to ADK
5. The agent execution resumes with the authenticated credentials

This approach allows ADK to handle complex authentication flows while keeping the user experience smooth and secure.

Authentication for Custom Tools

When building custom tools that require authentication, developers can implement authentication logic directly within the tool function. This approach is useful for tools that need to access protected resources but don't fit into the standard authentication patterns.

The tool function can: 1. Check if valid credentials exist in the session state 2. Request new credentials if needed 3. Use the credentials to access the protected resource 4. Store the credentials in the session state for future use

This flexibility allows developers to implement custom authentication flows while still benefiting from ADK's security and state management capabilities.

Sessions, Memory, and Runtime

Meaningful, multi-turn conversations require agents to understand context. Just like humans, they need to recall what's been said and done to maintain continuity and avoid repetition. The Agent Development Kit (ADK) provides structured ways to manage this context through `Session`, `State`, and `Memory`.

Introduction to Conversational Context

Think of interacting with your agent as having distinct **conversation threads**, potentially drawing upon **long-term knowledge**. ADK's context management system is designed to handle both immediate conversation context and longer-term knowledge retention.

Sessions: Tracking Individual Conversations

A `Session` is the ADK object designed specifically to track and manage individual conversation threads. Following the idea of a "conversation thread," just like you wouldn't start every text message from scratch, agents need context from the ongoing interaction.

The Session Object

When a user starts interacting with your agent, the `SessionService` creates a `Session` object (`google.adk.sessions.Session`). This object acts as the container holding everything related to that one specific chat thread. Here are its key properties:

1. **Identification (`id`, `app_name`, `user_id`)**: Unique labels for the conversation.
2. `id`: A unique identifier for this specific conversation thread, essential for retrieving it later.
3. `app_name`: Identifies which agent application this conversation belongs to.
4. `user_id`: Links the conversation to a particular user.
5. **History (`events`)**: A chronological sequence of all interactions (`Event` objects – user messages, agent responses, tool actions) that have occurred within this specific thread.
6. **Session Data (`state`)**: A place to store temporary data relevant only to this specific, ongoing conversation. This acts as the agent's "scratchpad" for the current interaction.

Managing Sessions with a SessionService

The `SessionService` handles the lifecycle of `Session` objects:

1. **Creating Sessions**: Initializes a new conversation thread with a unique ID.
2. **Retrieving Sessions**: Loads an existing conversation by its ID.
3. **Updating Sessions**: Adds new events or modifies state.
4. **Deleting Sessions**: Removes conversations that are no longer needed.

SessionService Implementations

ADK offers different implementations for the `SessionService`:

1. **InMemorySessionService**: Stores sessions in memory.
2. **Pros**: Simple, fast, no setup required.
3. **Cons**: All data is lost when the application restarts.
4. **Best for**: Local development and testing.
5. **Database-backed implementations**: Store sessions in persistent storage.
6. **Pros**: Data persists across application restarts.
7. **Cons**: Requires database setup and configuration.

8. **Best for**: Production environments.

9. **Cloud-based implementations**: Store sessions in cloud services.

10. **Pros**: Scalable, managed infrastructure.
11. **Cons**: Requires cloud service setup and configuration.
12. **Best for**: Production environments with high scalability needs.

State: The Session's Scratchpad

Within each `Session` (our conversation thread), the `state` attribute acts like the agent's dedicated scratchpad for that specific interaction. While `session.events` holds the full history, `session.state` is where the agent stores and updates dynamic details needed during the conversation.

What is session.state?

Conceptually, `session.state` is a dictionary holding key-value pairs. It's designed for information the agent needs to recall or track to make the current conversation effective:

- **Personalize Interaction**: Remember user preferences mentioned earlier (e.g., `"user_preference_theme": "dark"`).
- **Track Task Progress**: Keep tabs on steps in a multi-turn process (e.g., `"booking_step": "confirm_payment"`).
- **Accumulate Information**: Build lists or summaries (e.g., `"shopping_cart_items": ["book", "pen"]`).
- **Make Informed Decisions**: Store flags or values influencing the next response (e.g., `"user_is_authenticated": True`).

Key Characteristics of State

1. **Structure: Serializable Key-Value Pairs**
2. Data is stored as `key: value`.
3. Keys: Always strings (`str`). Use clear names (e.g., `"departure_city"`, `"user:language_preference"`).

4. Values: Must be serializable. This means primitive types (strings, numbers, booleans) or collections of these (lists, dictionaries).

5. **Scope: Session-Specific**

6. Data in `state` is only relevant to the current session.

7. When the session ends or is deleted, this data is typically lost (unless explicitly saved elsewhere).

8. **Persistence: Depends on SessionService**

9. With `InMemorySessionService`, state is lost when the application restarts.
10. With database or cloud implementations, state persists as long as the session exists.

Organizing State with Prefixes: Scope Matters

It's recommended to use prefixes in state keys to organize data by scope or ownership:

- **Tool-specific data**: `"tool_name.key"` (e.g., `"calendar.last_viewed_date"`)
- **Agent-specific data**: `"agent.key"` (e.g., `"agent.current_task"`)
- **User-specific data**: `"user.key"` (e.g., `"user.preferred_language"`)

How State is Updated: Recommended Methods

ADK provides several methods to update state safely:

1. **Using SessionService.update_session_state()**: `python session_service.update_session_state( session_id=session.id, updates={"key1": "new_value1", "key2": "new_value2"} )`

2. **Using Runner.update_session_state()**: `python runner.update_session_state( session_id=session.id, updates={"key1": "new_value1", "key2": "new_value2"} )`

3. **From within a Tool function**: ```python def my_tool_function(session): # Read from state current_value = session.state.get("my_key", "default")

   # Update state (will be committed after tool execution) session.state["my_key"] = "new_value" return "Tool result" ```

Warning About Direct State Modification

While direct modification of `session.state` is possible, it's important to understand when changes are committed:

- Changes made directly to `session.state` during tool execution are automatically committed when the tool completes.
- Changes made outside the normal execution flow (e.g., in custom code) need to be explicitly committed using one of the update methods.

Memory: Long-Term Knowledge with MemoryService

We've seen how `Session` tracks the history (`events`) and temporary data (`state`) for a single, ongoing conversation. But what if an agent needs to recall information from past conversations or access external knowledge bases? This is where the concept of Long-Term Knowledge and the `MemoryService` come into play.

Think of it this way: - `Session` / `State`: Like your short-term memory during one specific chat. - Long-Term Knowledge (`MemoryService`): Like a searchable archive or knowledge library the agent can consult, potentially containing information from many past chats or other sources.

The MemoryService Role

The `BaseMemoryService` defines the interface for managing this searchable, long-term knowledge store. Its primary responsibilities are:

1. **Ingesting Information (`add_session_to_memory`)**: Taking the contents of a (usually completed) `Session` and adding relevant information to the long-term knowledge store.
2. **Searching Information (`search_memory`)**: Allowing an agent (typically via a `Tool`) to query the knowledge store and retrieve relevant snippets or context based on a search query.

MemoryService Implementations

ADK offers different implementations for the `MemoryService`:

1. **InMemoryMemoryService**: Stores memory in RAM.
2. **Pros**: Simple, fast, no setup required.
3. **Cons**: All data is lost when the application restarts.
4. **Best for**: Local development and testing.
5. **Vector database implementations**: Store embeddings for semantic search.
6. **Pros**: Powerful semantic search capabilities.
7. **Cons**: Requires database setup and configuration.
8. **Best for**: Production environments requiring sophisticated memory retrieval.
9. **Cloud-based implementations**: Leverage cloud services for memory storage.
10. **Pros**: Scalable, managed infrastructure.

11. **Cons**: Requires cloud service setup and configuration.
12. **Best for**: Production environments with high scalability needs.

How Memory Works in Practice

The typical workflow for using memory in an agent application involves:

1. **Adding to Memory**: After a conversation completes, relevant information is extracted and added to the memory store. `python memory_service.add_session_to_memory(session)`

2. **Searching Memory**: During a conversation, the agent can search the memory to recall relevant information. `python results = memory_service.search_memory( query="What was the user's preferred shipping method?", limit=5 )`

3. **Using Memory Results**: The agent incorporates the retrieved information into its responses or decision-making process.

Runtime: The Engine Powering Agent Interactions

The ADK Runtime is the underlying engine that powers your agent application during user interactions. It's the system that takes your defined agents, tools, and callbacks and orchestrates their execution in response to user input, managing the flow of information, state changes, and interactions with external services like LLMs or storage.

Think of the Runtime as the "engine" of your agentic application. You define the parts (agents, tools), and the Runtime handles how they connect and run together to fulfill a user's request.

Core Idea: The Event Loop

At its heart, the ADK Runtime operates on an Event Loop. This loop facilitates a back-and-forth communication between the `Runner` component and your defined "Execution Logic" (which includes your Agents, the LLM calls they make, Callbacks, and Tools).

In simple terms:

1. The `Runner` receives a user query and asks the main `Agent` to start processing.
2. The `Agent` (and its associated logic) runs until it has something to report (like a response, a request to use a tool, or a state change) – it then `yields` an `Event`.

3. The `Runner` receives this `Event`, processes any associated actions (like saving state changes via `Services`), and forwards the event onwards (e.g., to the user interface).
4. Only `after` the `Runner` has processed the event does the `Agent`'s logic `resume` from where it paused, now potentially seeing the effects of the changes committed by the Runner.
5. This cycle repeats until the agent has no more events to yield for the current user query.

This event-driven loop is the fundamental pattern governing how ADK executes your agent code.

The Heartbeat: The Event Loop - Inner workings

The Event Loop is the core operational pattern defining the interaction between the `Runner` and your custom code (Agents, Tools, Callbacks, collectively referred to as "Execution Logic" or "Logic Components" in the design document). It establishes a clear division of responsibilities:

Runner's Role (Orchestrator)

The `Runner` is responsible for:

1. **Initializing the execution environment**: Setting up the session, state, and other resources.
2. **Processing events yielded by execution logic**: Taking appropriate actions based on event type.
3. **Committing state changes**: Ensuring state updates are properly saved.
4. **Managing the execution flow**: Deciding when to resume execution logic.
5. **Handling errors and exceptions**: Providing graceful error handling.

Execution Logic's Role (Agent, Tool, Callback)

Your execution logic (Agents, Tools, Callbacks) is responsible for:

1. **Processing user input**: Understanding and responding to user queries.
2. **Making decisions**: Determining what actions to take.
3. **Yielding events**: Communicating results, requests, and state changes to the Runner.
4. **Resuming execution**: Continuing from where it left off after the Runner processes events.

Key components of the Runtime

1. **Runner**: The orchestrator that manages the execution flow.
2. **Execution Logic Components**:
3. **Agent**: The main component that processes user input and generates responses.
4. **Tool**: Components that perform specific tasks (e.g., searching, calculating).
5. **Callback**: Components that are notified of specific events.
6. **Event**: The communication mechanism between the Runner and Execution Logic.
7. **Services**: Components that provide access to external resources (e.g., SessionService, MemoryService).
8. **Session**: The container for conversation state and history.
9. **Invocation**: A single execution of the Runtime in response to a user query.

How It Works: A Simplified Invocation

When a user sends a query to your agent application, the following steps occur:

1. The application calls `runner.run_async(session_id, user_id, new_message)`.
2. The Runner retrieves the session and prepares the execution environment.
3. The Runner adds the user's message as an event to the session.
4. The Runner invokes the main Agent with the updated session.
5. The Agent processes the input, potentially using LLMs and Tools.
6. When the Agent has a response or needs to use a Tool, it yields an Event.
7. The Runner processes the Event, updates the session, and resumes the Agent.
8. This cycle continues until the Agent has no more events to yield.
9. The Runner returns the final result to the application.

Important Runtime Behaviors

1. **State Updates & Commitment Timing**: State changes are only committed when the Runner processes events, not when they're made by execution logic.
2. **"Dirty Reads" of Session State**: Execution logic may see uncommitted state changes, which can lead to inconsistencies if not handled properly.
3. **Streaming vs. Non-Streaming Output**: The Runtime supports both streaming (partial=True) and non-streaming output modes.
4. **Async is Primary**: The primary API for running agents is asynchronous (`run_async`), which allows for better performance and responsiveness.

Artifacts

Artifacts in ADK are a way to store and manage binary data or large text content that doesn't fit well in the regular session state. They provide a mechanism for handling files, images, audio, or other binary data within the agent system.

Key characteristics of Artifacts:

1. **Binary Data Storage**: Designed for storing binary data or large text content.
2. **Separate from State**: Stored separately from session state to avoid performance issues.
3. **Reference-Based**: Referenced in session state via artifact IDs.
4. **Lifecycle Management**: Can be created, retrieved, and deleted.

Callbacks

Callbacks in ADK provide a way to hook into the agent execution lifecycle, allowing you to execute custom code at specific points during the agent's operation. They're useful for logging, monitoring, debugging, and extending the agent's functionality.

Key characteristics of Callbacks:

1. **Lifecycle Hooks**: Execute at specific points in the agent execution lifecycle.
2. **Custom Logic**: Allow you to add custom behavior without modifying the core agent code.
3. **Multiple Callbacks**: Multiple callbacks can be registered and will execute in registration order.
4. **Synchronous and Asynchronous**: Support both sync and async execution models.

Events

Events in ADK are the primary communication mechanism between the Runner and Execution Logic components. They represent discrete occurrences during agent execution, such as user messages, agent responses, tool calls, and state changes.

Key characteristics of Events:

1. **Communication Mechanism**: Used to communicate between Runner and Execution Logic.
2. **Typed Structure**: Each event has a specific type and structure.
3. **Chronological Record**: Stored in the session as a chronological record of the conversation.
4. **Yield-Based Flow**: Execution Logic yields events to the Runner for processing.

Context

Context in ADK refers to the information available to the agent during execution. It includes the session state, conversation history, and any other information needed to process user queries effectively.

Key aspects of Context:

1. **Information Aggregation**: Combines various sources of information for the agent.
2. **Dynamic Updates**: Updated throughout the conversation as new information becomes available.
3. **Scoped Access**: Different components may have access to different parts of the context.
4. **Structured Format**: Organized in a way that's easily consumable by the agent.

Evaluation, Guides, and API

Evaluation of Agents

In traditional software development, unit tests and integration tests provide confidence that code functions as expected and remains stable through changes. These tests provide a clear "pass/fail" signal, guiding further development. However, LLM agents introduce a level of variability that makes traditional testing approaches insufficient.

Due to the probabilistic nature of models, deterministic "pass/fail" assertions are often unsuitable for evaluating agent performance. Instead, qualitative evaluations of both the final output and the agent's trajectory (the sequence of steps taken to reach the solution) are needed. This involves assessing the quality of the agent's decisions, its reasoning process, and the final result.

Preparing for Agent Evaluations

Before automating agent evaluations, it's important to define clear objectives and success criteria:

1. **Define Success:** What constitutes a successful outcome for your agent?
2. **Identify Critical Tasks:** What are the essential tasks your agent must accomplish?
3. **Choose Relevant Metrics:** What metrics will you track to measure performance?

These considerations guide the creation of evaluation scenarios and enable effective monitoring of agent behavior in real-world deployments.

What to Evaluate?

Agent evaluation can be broken down into two components:

1. **Evaluating Trajectory and Tool Use:** Analyzing the steps an agent takes to reach a solution, including its choice of tools, strategies, and the efficiency of its approach.

2. **Evaluating the Final Response:** Assessing the quality, relevance, and correctness of the agent's final output.

The trajectory is the list of steps the agent took before it returned to the user. We can compare that against the list of steps we expect the agent to have taken.

Evaluating Trajectory and Tool Use

Before responding to a user, an agent typically performs a series of actions, which is referred to as a 'trajectory.' It might compare the user input with session history to disambiguate a term, look up a policy document, search a knowledge base, or invoke an API. Evaluating an agent's performance requires comparing its actual trajectory to an expected, or ideal, one.

Several ground-truth-based trajectory evaluations exist:

1. **Exact match:** Requires a perfect match to the ideal trajectory.
2. **In-order match:** Requires the correct actions in the correct order, allows for extra actions.
3. **Any-order match:** Requires the correct actions in any order, allows for extra actions.
4. **Precision:** Measures the relevance/correctness of predicted actions.
5. **Recall:** Measures how many essential actions are captured in the prediction.
6. **Single-tool use:** Checks for the inclusion of a specific action.

Choosing the right evaluation metric depends on the specific requirements and goals of your agent. For instance, in high-stakes scenarios, an exact match might be crucial, while in more flexible situations, an in-order or any-order match might suffice.

How Evaluation Works with the ADK

The ADK offers two methods for evaluating agent performance against predefined datasets and evaluation criteria:

First Approach: Using a Test File

This approach involves creating individual test files, each representing a single, simple agent-model interaction (a session). It's most effective during active agent development, serving as a form of unit testing. These tests are designed for rapid execution and should focus on simple session complexity.

Each test file contains a single session, which may consist of multiple turns. A turn represents a single interaction between the user and the agent. Each turn includes:

- `query` : The user query.

- `expected_tool_use`: The tool call(s) that we expect the agent to make to respond correctly to the user query.
- `expected_intermediate_agent_responses`: Natural language responses produced by the agent as it progresses towards a final answer.
- `reference`: The expected final response from the model.

Example test file format:

```
[
 {
  "query": "hi",
  "expected_tool_use": [],
  "expected_intermediate_agent_responses": [],
  "reference": "Hello! What can I do for you?\n"
 },
 {
  "query": "roll a die for me",
  "expected_tool_use": [
   {
    "tool_name": "roll_die",
    "tool_input": {
     "sides": 6
    }
   }
  ],
  "expected_intermediate_agent_responses": []
 }
]
```

Test files can be organized into folders. Optionally, a folder can also include a `test_config.json` file that specifies the evaluation criteria.

Second Approach: Using an Evalset File

The evalset approach utilizes a dedicated dataset called an "evalset" for evaluating agent-model interactions. Similar to a test file, the evalset contains example interactions. However, an evalset can contain multiple, potentially lengthy sessions, making it ideal for simulating complex, multi-turn conversations.

Due to its ability to represent complex sessions, the evalset is well-suited for integration tests. These tests are typically run less frequently than unit tests due to their more extensive nature.

Evaluation Criteria

The evaluation criteria define how the agent's performance is measured against the evalset. The following metrics are supported:

1. **`tool_trajectory_avg_score`**: This metric compares the agent's actual tool usage during the evaluation against the expected tool usage defined in the `expected_tool_use` field. Each matching tool usage step receives a score of 1, while a mismatch receives a score of 0. The final score is the average of these matches, representing the accuracy of the tool usage trajectory.

2. **`response_match_score`**: This metric compares the agent's final natural language response to the expected final response, stored in the `reference` field. The ROUGE metric is used to calculate the similarity between the two responses.

If no evaluation criteria are provided, the following default configuration is used: - `tool_trajectory_avg_score`: Defaults to 1.0, requiring a 100% match in the tool usage trajectory. - `response_match_score`: Defaults to 0.8, allowing for a small margin of error in the agent's natural language responses.

Example of a custom evaluation criteria configuration:

```
{
 "criteria": {
   "tool_trajectory_avg_score": 1.0,
   "response_match_score": 0.8
 }
}
```

How to Run Evaluation with the ADK

The ADK provides three methods for running evaluations:

1. adk web - Run Evaluations via the Web UI

The ADK web interface provides a user-friendly way to run and visualize evaluations. This approach is particularly useful for developers who prefer a graphical interface for monitoring and analyzing agent performance.

2. pytest - Run Tests Programmatically

For developers who prefer a programmatic approach, the ADK supports running evaluations using pytest. This method integrates well with existing CI/CD pipelines and allows for automated testing.

Example command:

```
python -m pytest path/to/test_file.py -v
```

Example test code:

```
from google.adk.evaluation import run_test_file

def test_agent():
    result = run_test_file(
        agent_factory=lambda: my_agent,
        test_file_path="path/to/test_file.json"
    )
    assert result.passed
```

3. adk eval - Run Evaluations via the CLI

The ADK command-line interface provides a convenient way to run evaluations from the terminal. This approach is particularly useful for quick evaluations during development.

Guides

Contributing Guide

The ADK project welcomes contributions to both the core Python framework and its documentation. The contributing guide provides information on how to get involved.

Repositories

1. **google/adk-python**: Contains the core Python library source code.
2. **google/adk-docs**: Contains the source for the documentation site.

Before You Begin

Before contributing, you need to:

1. **Sign the Contributor License Agreement (CLA)**: Contributions must be accompanied by a CLA. This gives Google permission to use and redistribute your contributions as part of the project.

2. **Review Community Guidelines**: Familiarize yourself with the project's community guidelines to ensure your contributions align with the project's goals and standards.

How to Contribute

There are several ways to contribute to the ADK project:

1. **Reporting Issues (Bugs & Errors)**: Help improve the project by reporting bugs and errors you encounter.
2. **Suggesting Enhancements**: Propose new features or improvements to existing functionality.
3. **Improving Documentation**: Help make the documentation more comprehensive, clear, and accurate.
4. **Writing Code**: Contribute code improvements, bug fixes, or new features.

Code Reviews

All submissions, including submissions by project members, require review. The project uses GitHub pull requests for this purpose.

API Reference

The ADK provides a comprehensive API for building, deploying, and evaluating agents. The API is organized into several modules:

Key Modules

1. **google.adk.agents**: Contains the core agent classes, including `BaseAgent` and `LlmAgent`.
2. **google.adk.artifacts**: Provides functionality for managing binary data or large text content.
3. **google.adk.code_executors**: Contains classes for executing code within agents.
4. **google.adk.evaluation**: Provides tools for evaluating agent performance.
5. **google.adk.events**: Contains classes for representing events in the agent execution lifecycle.
6. **google.adk.memory**: Provides functionality for managing long-term knowledge.
7. **google.adk.models**: Contains classes for interacting with language models.
8. **google.adk.planners**: Provides functionality for planning agent actions.

9. **google.adk.runners**: Contains the `Runner` class for orchestrating agent execution.
10. **google.adk.sessions**: Provides functionality for managing conversation sessions.
11. **google.adk.tools**: Contains various tool implementations for agents.

Key Classes

BaseAgent

The `BaseAgent` class is the foundation for all agents in the ADK. It provides the basic structure and functionality that all agents share.

Key properties and methods: - `name`: The name of the agent. - `description`: A description of the agent's purpose and capabilities. - `parent_agent`: The parent agent, if this agent is a sub-agent. - `sub_agents`: A list of sub-agents. - `run_async()`: Runs the agent asynchronously. - `run_live()`: Runs the agent with live updates. - `find_agent()`: Finds an agent by name. - `find_sub_agent()`: Finds a sub-agent by name.

LlmAgent

The `LlmAgent` class extends `BaseAgent` to provide functionality specific to language model-based agents.

Key properties and methods: - `model`: The language model to use. - `tools`: The tools available to the agent. - `instruction`: The instruction to give to the model. - `global_instruction`: A global instruction that applies to all interactions. - `examples`: Example interactions to guide the model. - `planner`: The planner to use for action planning. - `code_executor`: The code executor to use for running code. - `input_schema`: The schema for input data. - `output_schema`: The schema for output data. - `generate_content_config`: Configuration for content generation.

GitHub Repository Structure

The ADK is organized into two main repositories:

1. **google/adk-python**: Contains the core Python library, including:
2. Source code for all ADK modules
3. Tests for verifying functionality
4. Examples demonstrating usage patterns
5. Documentation comments within the code
6. **google/adk-docs**: Contains the documentation site, including:

7. Conceptual guides explaining ADK concepts
8. Tutorials for getting started
9. API reference documentation
10. Examples and best practices

Conclusion

The Google Agent Development Kit (ADK) represents a significant advancement in the field of AI agent development. By providing a comprehensive framework for building, testing, and deploying agents, it addresses many of the challenges that developers face when creating sophisticated AI applications.

Key Takeaways

1. **Unified Framework**: The ADK offers a cohesive approach to agent development, integrating language models, tools, and context management into a single framework.

2. **Flexible Architecture**: With support for different agent types, deployment options, and integration methods, the ADK can accommodate a wide range of use cases and requirements.

3. **Robust Context Management**: The session, state, and memory components provide a sophisticated system for managing conversational context, enabling agents to maintain coherent, multi-turn interactions.

4. **Comprehensive Tooling**: The ADK's extensive tool ecosystem allows agents to perform a variety of tasks, from simple calculations to complex API interactions.

5. **Evaluation and Testing**: The built-in evaluation framework helps ensure agent quality and reliability, with support for both unit testing and integration testing.

6. **Production-Ready**: With deployment options like Agent Engine and Cloud Run, the ADK supports the full lifecycle from prototype to production.

Future Directions

As AI technology continues to evolve, the ADK is likely to expand in several directions:

1. **Enhanced Model Support**: Integration with new and improved language models as they become available.

2. **Additional Tool Types**: Expansion of the tool ecosystem to support more specialized use cases.
3. **Improved Evaluation Methods**: Development of more sophisticated evaluation techniques to better assess agent performance.
4. **Advanced Context Management**: Further refinement of the context management system to handle more complex conversational scenarios.
5. **Expanded Deployment Options**: Support for additional deployment environments and integration with more cloud platforms.

Final Thoughts

The Google Agent Development Kit represents a significant step forward in making AI agent development more accessible and efficient. By providing a structured framework, comprehensive documentation, and a supportive community, it enables developers to create sophisticated AI applications that can understand and respond to natural language, use tools to accomplish tasks, and maintain context across conversations.

Whether you're building a simple chatbot or a complex multi-agent system, the ADK provides the tools and patterns you need to succeed. As the field of AI continues to advance, the ADK will likely evolve to incorporate new capabilities and best practices, further enhancing its value to the developer community.