```
# Initial setup (common for many examples)
from google.adk.agents import Agent, LlmAgent, BaseAgent, SequentialAgent,
ParallelAgent, LoopAgent
from google.adk.tools import FunctionTool, google_search
from google.adk.runners import Runner
from google.adk.sessions import InMemorySessionService
from google.genai import types
import datetime
from zoneinfo import ZoneInfo
from google.adk.tools.langchain_tool import LangchainTool
from langchain_community.tools import TavilySearchResults
from google.adk.tools.crewai_tool import CrewaiTool
from crewai_tools import SerperDevTool
from google.adk.tools.mcp_tool import MCPToolset, StdioServerParameters
import asyncio
import json
from google.adk.tools.openapi_tool import OpenAPIToolset
from pydantic import BaseModel, Field
from google.adk.agents.callback_context import CallbackContext
from google.adk.models import LlmResponse, LlmRequest
from google.adk.artifacts.in_memory_artifact_service import InMemoryArtifactService
```

Agent Development Kit

The Google Agent Development Kit (ADK) is an open-source Python toolkit designed to help developers build, evaluate, and deploy sophisticated AI agents with flexibility and control. It provides a code-first approach, allowing developers to define agent behavior, orchestration, and tool use directly in Python.

Agents

In the ADK, an Agent is a self-contained execution unit that acts autonomously to achieve specific goals. Agents can perform tasks, interact with users, utilize external tools, and coordinate with other agents. The foundation for all agents in ADK is the `BaseAgent` class.

LLM agents

LLM Agents (`LlmAgent` or simply `Agent`) utilize Large Language Models (LLMs) as their core engine to understand natural language, reason, plan, generate responses, and dynamically decide how to proceed or which tools to use. They are ideal for flexible, language-centric tasks.

Python
```
# Example: Defining a basic LLM agent
capital_agent = LlmAgent(
   model="gemini-2.0-flash-exp",
   name="capital_agent",
   description="Answers...[source](https://google.github.io/adk-docs/agents/llm-agents/)
)
```

Workflow agents

Workflow Agents control the execution flow of other agents in predefined, deterministic patterns without using an LLM for the flow control itself. They are perfect for structured processes needing predictable execution. There are three main types:

Sequential agents: Execute a series of sub-agents in a fixed, strict order.

```python
 Python
# Example: Sequential agent to get page content and then summarize it
class ContentGetter(LlmAgent):
    def __init__(self, **kwargs):
        super().__init__(name="ContentGetter", instruction="Get the content of the webpage.",
**kwargs)

class Summarizer(LlmAgent):
    def __init__(self, **kwargs):
        super().__init__(name="Summarizer", instruction="Summarize the content.", **kwargs)

page_content_getter = ContentGetter(model="gemini-2.0-flash-exp")
page_summarizer = Summarizer(model="gemini-2.0-flash-exp")

summarization_workflow = SequentialAgent(
    name="WebpageSummarizer",
    sub_agents=[page_content_getter, page_summarizer]
)
```

-
- **Loop agents:** Execute a sub-agent repeatedly based on a condition. (Example code for Loop agents was not explicitly provided in the research snippets).

- **Parallel agents:** Execute multiple sub-agents simultaneously. (Example code for Parallel agents was not explicitly provided in the research snippets).

Custom agents

Custom Agents are created by extending `BaseAgent` directly, allowing you to implement unique operational logic, specific control flows, or specialized integrations not covered by the standard types.

```python
Python
# Example: Simplified custom agent initialization (Conceptual from documentation)
class StoryFlowAgent(BaseAgent):
    """Custom agent for a story generation and refinement workflow."""
    def __init__(self, story_generator, story_critiquer, grammar_checker, tone_checker,
**kwargs):
        super().__init__(sub_agents=[story_generator, story_critiquer, grammar_checker,
tone_checker], **kwargs)
        self.story_generator = story_generator
        self.story_critiquer = story_critiquer
        self.grammar_checker = grammar_checker
        self.tone_checker = tone_checker

    async def _run_async_impl(self, ctx):
        # Define custom execution logic here (simplified for example)
        async for event in self.story_generator.run_async(ctx):
            yield event
        async for event in self.story_critiquer.run_async(ctx):
            yield event
        #... more steps
```

Multi-agent systems

Multi-Agent Systems involve combining different agent types to create sophisticated, collaborative systems capable of tackling complex problems. This often involves LLM Agents for intelligent tasks, Workflow Agents for managing process flow, and Custom Agents for specialized capabilities.

Python
```
# Example: Conceptual multi-agent system with coordinator, booking agent, and info agent
booking_agent = LlmAgent(name="Booker", description="Handles flight and hotel
bookings.", model="gemini-2.0-flash-exp")
info_agent = LlmAgent(name="Info", description="Provides general information and answers
questions.", model="gemini-2.0-flash-exp")
coordinator = LlmAgent(
    name="Coordinator",
    instruction="You are an assistant. Delegate booking tasks to Booker and info requests to
Info.",
    description="Main coordinator.",
    sub_agents=[booking_agent, info_agent],
    model="gemini-2.0-flash-exp"
)
```

Models

The ADK is designed for flexibility, allowing you to integrate various Large Language Models (LLMs) into your agents. It supports Google Gemini models, models from Vertex AI Model Garden, and models from other providers via LiteLLM.

Python
```
# Example: Using a Google Gemini model
```

```
gemini_agent = LlmAgent(model="gemini-2.0-flash-exp", name="gemini_agent",
instruction="Answer questions.")

# Example: Using a model via LiteLLM (conceptual - requires LiteLLM setup)
# from google.adk.models.litellm import LiteLlm
# litellm_model = LiteLlm(model_name="openai/gpt-4")
# openai_agent = LlmAgent(model=litellm_model, name="openai_agent",
instruction="Answer questions.")
```

Tools

Tools extend the capabilities of AI agents, allowing them to interact with the real world, access information, and perform specific actions.

Function tools

Function Tools allow you to integrate custom Python functions as tools that your agents can call.

```
Python
# Example: Defining a function tool to get weather report
def get_weather_report(city: str) -> dict:
    """Retrieves the current weather report for a specified city."""
    if city.lower() == "london":
        return {"status": "success", "report": "The current weather in London is cloudy..."}
    elif city.lower() == "paris":
        return {"status": "success", "report": "The weather in Paris is sunny..."}
    else:
        return {"status": "error", "error_message": f"Weather information for '{city}' is not
available."}

weather_tool = FunctionTool(func=get_weather_report)

# Integrating the tool into an agent
weather_sentiment_agent = LlmAgent(
    name="weather_sentiment_agent",
    model="gemini-2.0-flash-exp",
    instruction="You can get the weather report for a city.",
    tools=[weather_tool]
)
```

Built-in tools

Built-in Tools are pre-packaged tools that offer agents immediate access to common functionalities.

Python

```
# Example: Using the built-in Google Search tool
search_agent = Agent(
    name="basic_search_agent",
    model="gemini-2.0-flash-exp",
    instruction="Just ask me anything!",
    tools=[google_search] # google_search is a pre-built tool
)
```

Third party tools

The ADK allows for the integration of various third-party tools and services, such as those from LangChain and CrewAI.

Python
```
# Example: Integrating a LangChain tool (Tavily Search)
# Ensure dependencies are installed: pip install langchain_community tavily-python
import os
os.environ = "<REPLACE_WITH_API_KEY>" # Replace with your actual API key

tavily_tool_instance = TavilySearchResults(max_results=5)
adk_tavily_tool = LangchainTool(tool=tavily_tool_instance)

langchain_agent = Agent(
    name="langchain_tool_agent",
    model="gemini-2.0-flash-exp",
    description="Agent to answer questions using TavilySearch.",
    instruction="I can answer your questions by searching the internet. Just ask me
anything!",
    tools=[adk_tavily_tool]
)

# Example: Integrating a CrewAI tool (Serper API)
# Ensure dependencies are installed: pip install crewai-tools
os.environ = "<REPLACE_WITH_API_KEY>" # Replace with your actual API key

serper_tool_instance = SerperDevTool(n_results=10)
adk_serper_tool = CrewaiTool(name="InternetNewsSearch", description="Searches the
internet for news.", tool=serper_tool_instance)

crewai_agent = Agent(
    name="crewai_search_agent",
    model="gemini-2.0-flash-exp",
    description="Agent to find recent news using the Serper search tool.",
    instruction="I can find the latest news for you. What topic are you interested in?",
    tools=[adk_serper_tool]
)
```

Google Cloud tools

Google Cloud Tools facilitate connecting your agents to various Google Cloud products and services. Examples include integration with Apigee APIs, Integration Connectors, Application Integration workflows, and databases using the MCP Toolbox. (Specific code examples for various Google Cloud tools require specific Google Cloud setup and configurations, which are detailed in the documentation).

MCP tools

MCP (Model Context Protocol) Tools allow agents to connect to a vast and diverse range of data sources and existing capabilities through the MCP standard.

```
Python
# Example: Connecting to a local File System MCP Server (Conceptual)
async def mcp_example():
    tools, exit_stack = await MCPToolset.from_server(
        connection_params=StdioServerParameters(
            command='npx',
            args=["-y", "@modelcontextprotocol/server-filesystem", "/path/to/your/folder"], #
Replace with your path
        )
    ).get_tools_async()

    root_agent = LlmAgent(
        model='gemini-2.0-flash',
        name='mcp_filesystem_agent',
        instruction="Use the available tools to interact with the file system."
    )
    #... (rest of the agent setup and running logic)

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

OpenAPI tools

OpenAPI Tools enable integration with external REST APIs by automatically generating callable tools directly from an OpenAPI Specification (v3.x).

```
Python
# Example: Integrating with an OpenAPI specification (Conceptual)
openapi_spec_url = "https://petstore.swagger.io/v2/swagger.json" # Example URL
openapi_toolset = OpenAPIToolset.from_url(openapi_spec_url)
openapi_tools = openapi_toolset.get_tools()

openapi_agent = LlmAgent(
    model="gemini-2.0-flash-exp",
    name="openapi_agent",
    instruction="Use the available API tools to answer user questions.",
    tools=openapi_tools
)
```

Authentication

The ADK provides mechanisms to ensure the security of agent interactions and control access to resources. This includes setting up credentials for Google Cloud services and handling user authentication using protocols like OAuth 2.0. (Detailed authentication implementation often involves specific cloud configurations and client-side integrations, as described in the documentation).

Deploy

The ADK offers flexible deployment options for your agents.

Agent Engine

Agent Engine is a fully managed, auto-scaling service on Google Cloud's Vertex AI specifically designed for deploying, managing, and scaling AI agents built with frameworks like ADK.

Python
```python
# Example: Preparing an agent for Agent Engine deployment
from vertexai.preview import reasoning_engines

def get_weather(city: str) -> dict:
    #... (weather retrieval logic)

def get_current_time(city: str) -> dict:
    #... (time retrieval logic)

root_agent = Agent(
    name="weather_time_agent",
    model="gemini-2.0-flash-exp",
    description="Agent to answer questions about time and weather.",
    instruction="I can answer your questions about the time and weather in a city.",
    tools=[get_weather, get_current_time]
)

app = reasoning_engines.AdkApp(
    agent=root_agent,
    enable_tracing=True,
)
```

Cloud Run

Cloud Run is a managed auto-scaling compute platform on Google Cloud that enables you to run your agent as a container-based application. You can deploy using the `adk deploy cloud_run` command or with `gcloud run deploy` using a Dockerfile.

```
Bash
# Example: Minimal command to deploy to Cloud Run using ADK CLI
adk deploy cloud_run \
  --project=$GOOGLE_CLOUD_PROJECT \
  --region=$GOOGLE_CLOUD_LOCATION \
 ./your_agent_directory # Replace with the path to your agent code
```

Sessions & Memory

The ADK provides structured ways to manage conversational context.

Session

A Session represents a single, ongoing interaction between a user and your agent system. It contains the chronological sequence of messages and actions (Events) and can hold temporary data (State) relevant only during this conversation.

```
Python
# Example: Setting up a session service and creating a session
session_service = InMemorySessionService()
APP_NAME = "my_app"
USER_ID = "user_1"
SESSION_ID = "session_001"
session = session_service.create_session(app_name=APP_NAME, user_id=USER_ID,
session_id=SESSION_ID)
```

State

State refers to the data stored within a specific Session. It's used to manage information relevant only to the current, active conversation thread.

```
Python
# Example: Reading and writing to session state within a tool function
def my_tool_function(tool_context):
    # Reading state
    previous_query = tool_context.state.get("previous_query")
    print(f"Previous query: {previous_query}")

    # Writing state
    tool_context.state["current_result"] = "Some result"
    return {"result": "Operation successful"}
```

Memory

Memory represents a store of information that might span multiple past sessions or include external data sources. It acts as a knowledge base the agent can search to recall information beyond the immediate conversation. (The research snippets primarily mention the concept of memory and its management by `MemoryService`, but don't provide specific code examples for different memory implementations).

Artifacts

Artifacts refer to the management of various forms of data and files associated with agent interactions, such as file uploads and downloads. While the outline mentions "None," the ADK does support handling artifacts.

Python
```
# Example: Saving an artifact within a callback (Conceptual)
async def after_report_generation(callback_context: CallbackContext, response:
types.Content):
    report_content = response.parts.text
    await callback_context.save_artifact("report.txt", report_content.encode('utf-8'))
```

Callbacks

Callbacks are standard Python functions that you define and associate with an agent to observe, customize, and even control the agent's behavior at specific points in its lifecycle.

Types of callbacks

The ADK provides different types of callbacks that trigger at various stages of an agent's execution:

- **Agent Lifecycle Callbacks:** `before_agent_callback`, `after_agent_callback`.
- **LLM Interaction Callbacks:** `before_model_callback`, `after_model_callback`.
- **Tool Execution Callbacks:** `before_tool_callback`, `after_tool_callback`.
-

Python
```
# Example: before_model_callback to add dynamic instructions
def add_context_to_prompt(callback_context: CallbackContext, llm_request: LlmRequest) ->
None:
    user_tier = callback_context.state.get("user_tier", "basic")
    if user_tier == "premium":
        llm_request.contents.parts.text = f"As a premium user,
{llm_request.contents.parts.text}"
```

```
premium_agent = LlmAgent(
    model="gemini-2.0-flash-exp",
    name="premium_support_agent",
    instruction="Provide helpful support.",
    callbacks=[add_context_to_prompt]
)
```

Callback patterns

Callback patterns are common ways to use callbacks to achieve specific functionalities, such as implementing guardrails, dynamic state management, logging, caching, and request/response modification. (Examples of callback patterns are integrated into the "Types of callbacks" section above).

Runtime

The ADK Runtime is the underlying engine that powers your agent application during user interactions. It orchestrates the execution of agents, tools, and callbacks in response to user input, managing the flow of information, state changes, and interactions with external services. The runtime operates on an Event Loop.

Events

Events are the fundamental units of information flow within the ADK. They represent every significant occurrence during an agent's interaction lifecycle, from user input to the final response and all the steps in between.

Python
```
# Example: Conceptual event structure (not directly created by user in this form)
event = types.Event(
    author="user",
    content=types.Content(parts=[types.Part(text="What's the weather in London?")]),
    session_id="session_123",
    user_id="user_456"
)
```

Context

Context refers to the crucial bundle of information available to your agent and its tools during specific operations. It includes the user's input, conversation history, agent state, and environmental variables. The `InvocationContext` acts as the comprehensive internal container for this information.

Python
```
# Example: Accessing context within a tool function
def another_tool_function(tool_context):
    user_id = tool_context.user_id
    session_id = tool_context.session_id
    current_state = tool_context.state
    print(f"User ID: {user_id}, Session ID: {session_id}, State: {current_state}")
    return {"info": "Context accessed"}
```

Evaluate

The ADK provides tools and methods to systematically evaluate the performance and effectiveness of your AI agents. This includes evaluating the quality of the final response and the step-by-step execution trajectory. Evaluation can be done via a web-based UI (`adk web`), programmatically using `pytest`, or via the command-line interface (`adk eval`).

Python
```
# Example: Programmatic evaluation using pytest (Conceptual)
# In a test file (e.g., tests/integration/test_agent.py)
# from google.adk.evals import AgentEvaluator

# def test_capital_agent():
#     AgentEvaluator.evaluate(
#         agent_module="path.to.your.capital_agent_module",
#         eval_dataset="path/to/your/evaluation_dataset.json"
#     )
```

This detailed explanation with code examples should provide a comprehensive understanding of the Google Agent Development Kit and its various components. Remember to refer to the official ADK documentation for the most up-to-date information and more advanced use cases.