Google Agent Development Kit (ADK): Agent Types and Models

Agent Types Overview

The Agent Development Kit (ADK) provides several distinct agent categories to build sophisticated applications. The foundation for all agents in ADK is the `BaseAgent` class, which serves as the fundamental blueprint. To create functional agents, developers typically extend `BaseAgent` in one of three main ways, catering to different needs – from intelligent reasoning to structured process control.

1. LLM Agents (`LlmAgent`, `Agent`)

LLM Agents 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.

Key characteristics: - Leverage the power of a Large Language Model (LLM) for reasoning, understanding natural language, making decisions, generating responses, and interacting with tools - Behavior is non-deterministic, as the LLM dynamically decides how to proceed, which tools to use, or whether to transfer control to another agent - Require configuration of identity, instructions, and tools

Configuration parameters: - `name` (Required): A unique string identifier for the agent - `model` (Required): Specifies the underlying LLM that will power the agent's reasoning - `description` (Optional, Recommended for Multi-Agent): A concise summary of the agent's capabilities - `instruction` (Optional but critical): A string that tells the agent its core task, personality, constraints, and how to use its tools - `tools` (Optional): A list of tools the agent can use, which can be Python functions, instances of classes inheriting from `BaseTool`, or other agents

Example:

```
 from google.adk.agents import LlmAgent
from google.adk.tools import google_search

dice_agent = LlmAgent(
    model="gemini-2.0-flash-exp",  # Required: Specify the LLM
    name="question_answer_agent",  # Required: Unique agent name
```

```
    description="A helpful assistant agent that can answer questions.",
    instruction="""Respond to the query using google search""",
    tools=[google_search],  # Provide an instance of the tool
)
```

2. Workflow Agents (`SequentialAgent`, `ParallelAgent`, `LoopAgent`)

Workflow Agents are specialized components designed purely for orchestrating the execution flow of sub-agents. They control the execution flow of other agents in predefined, deterministic patterns (sequence, parallel, or loop) without using an LLM for the flow control itself. They are perfect for structured processes needing predictable execution.

Key characteristics: - Operate based on predefined logic, not LLM reasoning - Determine execution sequence according to their type - Result in deterministic and predictable execution patterns - Primary role is to manage how and when other agents run

Types of Workflow Agents:

1. **Sequential Agents**: Execute sub-agents one after another, in sequence
2. **Loop Agents**: Repeatedly execute sub-agents until a specific termination condition is met
3. **Parallel Agents**: Execute multiple sub-agents in parallel

Benefits of Workflow Agents: - Predictability: The flow of execution is guaranteed based on the agent type and configuration - Reliability: Ensures tasks run in the required order or pattern consistently - Structure: Allows building complex processes by composing agents within clear control structures

3. Custom Agents

Custom Agents are created by extending `BaseAgent` directly, allowing developers to implement unique operational logic, specific control flows, or specialized integrations not covered by the standard types. They cater to highly tailored application requirements.

Key characteristics: - Provide the ultimate flexibility in ADK by allowing arbitrary orchestration logic - Created by inheriting directly from `BaseAgent` and implementing custom control flow - Go beyond the predefined patterns of workflow agents - Enable highly specific and complex agentic workflows

Implementation approach: - Inherit from `BaseAgent` - Implement the `_run_async_impl` method to define custom execution logic - Manage sub-agents and state as needed

Example use cases: - Custom decision trees that don't fit standard sequential/parallel/loop patterns - Specialized state machines with complex transitions - Integration with external systems requiring custom logic

Multi-Agent Systems

In ADK, a multi-agent system is an application where different agents, often forming a hierarchy, collaborate or coordinate to achieve a larger goal. As agentic applications grow in complexity, structuring them as a single, monolithic agent can become challenging to develop, maintain, and reason about.

Key components for agent composition:

1. **Agent Hierarchy**: The parent-child relationship defined in `BaseAgent` where:
2. You create a tree structure by passing a list of agent instances to the `sub_agents` argument
3. Each agent can only have one parent (Single Parent Rule)
4. The hierarchy defines the scope for workflow agents and influences potential targets for LLM-driven delegation
5. **Interaction & Communication Mechanisms**:
6. Shared Session State: Agents can store and retrieve data from `session.state`
7. LLM-Driven Delegation: Agents can transfer control to other agents based on LLM decisions
8. Explicit Invocation: Agents can be used as tools by other agents through `AgentTool`

Common Multi-Agent Patterns: - Coordinator/Dispatcher Pattern - Sequential Pipeline Pattern - Parallel Fan-Out/Gather Pattern - Hierarchical Task Decomposition - Review/Critique Pattern (Generator-Critic) - Iterative Refinement Pattern - Human-in-the-Loop Pattern

Models

The Agent Development Kit (ADK) is designed for flexibility, allowing integration with various Large Language Models (LLMs) into agents. While Google Gemini models are tightly integrated, ADK supports a wide range of models from different providers.

Model integration mechanisms:

1. **Direct String / Registry**: For models tightly integrated with Google Cloud (like Gemini models accessed via Google AI Studio or Vertex AI). Developers provide the model name or endpoint resource string directly to the `LlmAgent`.
2. **Wrapper Classes**: For broader compatibility, especially with models outside the Google ecosystem or those requiring specific client configurations (like models accessed via LiteLLM). Developers instantiate a specific wrapper class and pass this object as the `model` parameter.

Supported model types:

1. **Google Gemini Models**: The most direct way to use Google's flagship models within ADK.
2. **Cloud & Proprietary Models via LiteLLM**: Integration with models from providers like Anthropic, Meta, Mistral AI, AI21 Labs, and others.
3. **Open & Local Models via LiteLLM**: Support for open-source models running locally or on custom endpoints.
4. **Ollama Integration**: Support for running open-source models locally using Ollama.
5. **Self-Hosted Endpoints**: Support for models deployed on custom endpoints, such as those using vLLM.
6. **Models on Vertex AI**:
7. Model Garden Deployments
8. Fine-tuned Model Endpoints
9. Third-Party Models on Vertex AI (e.g., Anthropic Claude)

This flexibility allows developers to choose the most appropriate model for their specific use case, balancing factors like performance, cost, and deployment requirements.