LLM agents - Agent Development Kit

Source URL: https://google.github.io/adk-docs/agents/llm-agents/

LLM Agent¶

The `LlmAgent` (often aliased simply as `Agent`) is a core component in ADK, acting as the "thinking" part of your application. It leverages the power of a Large Language Model (LLM) for reasoning, understanding natural language, making decisions, generating responses, and interacting with tools.

Unlike deterministic Workflow Agents that follow predefined execution paths, `LlmAgent` behavior is non-deterministic. It uses the LLM to interpret instructions and context, deciding dynamically how to proceed, which tools to use (if any), or whether to transfer control to another agent.

Building an effective `LlmAgent` involves defining its identity, clearly guiding its behavior through instructions, and equipping it with the necessary tools and capabilities.

Defining the Agent's Identity and Purpose¶

First, you need to establish what the agent *is* and what it's *for*.

- **`name` (Required):** Every agent needs a unique string identifier. This `name` is crucial for internal operations, especially in multi-agent systems where agents need to refer to or delegate tasks to each other. Choose a descriptive name that reflects the agent's function (e.g., `customer_support_router`, `billing_inquiry_agent`). Avoid reserved names like `user`.
- **`description` (Optional, Recommended for Multi-Agent):** Provide a concise summary of the agent's capabilities. This description is primarily used by *other* LLM agents to determine if they should route a task to this agent. Make it specific enough to differentiate it from peers (e.g., "Handles inquiries about current billing statements," not just "Billing agent").

- **`model` (Required):** Specify the underlying LLM that will power this agent's reasoning. This is a string identifier like `"gemini-2.0-flash"`. The choice of model impacts the agent's capabilities, cost, and performance. See the Models page for available options and considerations.

PythonJava

```
# Example: Defining the basic identity
capital_agent = LlmAgent(
    model="gemini-2.0-flash",
    name="capital_agent",
    description="Answers user questions about the capital city of a gi
    # instruction and tools will be added next
)
```

```
// Example: Defining the basic identity
LlmAgent capitalAgent =
    LlmAgent.builder()
        .model("gemini-2.0-flash")
        .name("capital_agent")
        .description("Answers user questions about the capital city of
        // instruction and tools will be added next
        .build();
```

Guiding the Agent: Instructions (`instruction`)¶

The `instruction` parameter is arguably the most critical for shaping an `LlmAgent`'s behavior. It's a string (or a function returning a string) that tells the agent:

- Its core task or goal.
- Its personality or persona (e.g., "You are a helpful assistant," "You are a witty pirate").

- Constraints on its behavior (e.g., "Only answer questions about X," "Never reveal Y").
- How and when to use its `tools`. You should explain the purpose of each tool and the circumstances under which it should be called, supplementing any descriptions within the tool itself.
- The desired format for its output (e.g., "Respond in JSON," "Provide a bulleted list").

Tips for Effective Instructions:

- **Be Clear and Specific:** Avoid ambiguity. Clearly state the desired actions and outcomes.
- **Use Markdown:** Improve readability for complex instructions using headings, lists, etc.
- **Provide Examples (Few-Shot):** For complex tasks or specific output formats, include examples directly in the instruction.
- **Guide Tool Use:** Don't just list tools; explain *when* and *why* the agent should use them.

State:

- The instruction is a string template, you can use the `{var}` syntax to insert dynamic values into the instruction.
- `{var}` is used to insert the value of the state variable named var.
- `{artifact.var}` is used to insert the text content of the artifact named var.
- If the state variable or artifact does not exist, the agent will raise an error. If you want to ignore the error, you can append a `?` to the variable name as in `{var?}`.

PythonJava

```
 # Example: Adding instructions
capital_agent = LlmAgent(
    model="gemini-2.0-flash",
    name="capital_agent",
    description="Answers user questions about the capital city of a gi
    instruction="""You are an agent that provides the capital city of
When a user asks for the capital of a country:
```

```
1. Identify the country name from the user's query.
2. Use the `get_capital_city` tool to find the capital.
3. Respond clearly to the user, stating the capital city.
Example Query: "What's the capital of {country}?"
Example Response: "The capital of France is Paris."
""",
    # tools will be added next
)
```

```
// Example: Adding instructions
LlmAgent capitalAgent =
    LlmAgent.builder()
        .model("gemini-2.0-flash")
        .name("capital_agent")
        .description("Answers user questions about the capital city of
        .instruction(
            """
            You are an agent that provides the capital city of a count
            When a user asks for the capital of a country:
            1. Identify the country name from the user's query.
            2. Use the `get_capital_city` tool to find the capital.
            3. Respond clearly to the user, stating the capital city.
            Example Query: "What's the capital of {country}?"
            Example Response: "The capital of France is Paris."
            """)
        // tools will be added next
        .build();
```

(Note: For instructions that apply to all *agents in a system, consider using* `global_instruction` *on the root agent, detailed further in the Multi-Agents section.)*

Equipping the Agent: Tools (`tools`)¶

Tools give your `LlmAgent` capabilities beyond the LLM's built-in knowledge or reasoning. They allow the agent to interact with the outside world, perform calculations, fetch real-time data, or execute specific actions.

- **`tools` (Optional):** Provide a list of tools the agent can use. Each item in the list can be:
- A native function or method (wrapped as a `FunctionTool`). Python ADK automatically wraps the native function into a `FuntionTool` whereas, you must explicitly wrap your Java methods using `FunctionTool.create(...)`
- An instance of a class inheriting from `BaseTool`.
- An instance of another agent (`AgentTool`, enabling agent-to-agent delegation - see Multi-Agents).

The LLM uses the function/tool names, descriptions (from docstrings or the `description` field), and parameter schemas to decide which tool to call based on the conversation and its instructions.

PythonJava

```
 # Define a tool function
def get_capital_city(country: str) -> str:
  """Retrieves the capital city for a given country."""
  # Replace with actual logic (e.g., API call, database lookup)
  capitals = {"france": "Paris", "japan": "Tokyo", "canada": "Ottawa"}
  return capitals.get(country.lower(), f"Sorry, I don't know the capit

# Add the tool to the agent
capital_agent = LlmAgent(
    model="gemini-2.0-flash",
    name="capital_agent",
    description="Answers user questions about the capital city of a gi
    instruction="""You are an agent that provides the capital city of
    tools=[get_capital_city] # Provide the function directly
```

```
)
```

```
// Define a tool function
// Retrieves the capital city of a given country.
public static Map<String, Object> getCapitalCity(
        @Schema(name = "country", description = "The country to get ca
        String country) {
  // Replace with actual logic (e.g., API call, database lookup)
  Map<String, String> countryCapitals = new HashMap<>();
  countryCapitals.put("canada", "Ottawa");
  countryCapitals.put("france", "Paris");
  countryCapitals.put("japan", "Tokyo");

  String result =
          countryCapitals.getOrDefault(
                  country.toLowerCase(), "Sorry, I couldn't find the c
  return Map.of("result", result); // Tools must return a Map
}

// Add the tool to the agent
FunctionTool capitalTool = FunctionTool.create(experiment.getClass(),
LlmAgent capitalAgent =
    LlmAgent.builder()
        .model("gemini-2.0-flash")
        .name("capital_agent")
        .description("Answers user questions about the capital city of
        .instruction("You are an agent that provides the capital city
        .tools(capitalTool) // Provide the function wrapped as a Funct
        .build();
```

Learn more about Tools in the Tools section.

Advanced Configuration & Control¶

Beyond the core parameters, `LlmAgent` offers several options for finer control:

Fine-Tuning LLM Generation (`generate_content_config`)¶

You can adjust how the underlying LLM generates responses using `generate_content_config`.

- **`generate_content_config` (Optional):** Pass an instance of `google.genai.types.GenerateContentConfig` to control parameters like `temperature` (randomness), `max_output_tokens` (response length), `top_p`, `top_k`, and safety settings.

PythonJava

```
from google.genai import types

agent = LlmAgent(
    # ... other params
    generate_content_config=types.GenerateContentConfig(
        temperature=0.2, # More deterministic output
        max_output_tokens=250
    )
)
```

```
import com.google.genai.types.GenerateContentConfig;

LlmAgent agent =
    LlmAgent.builder()
        // ... other params
        .generateContentConfig(GenerateContentConfig.builder()
            .temperature(0.2F) // More deterministic output
            .maxOutputTokens(250)
            .build())
```

```
        .build();
```

Structuring Data (`input_schema`, `output_schema`, `output_key`)¶

For scenarios requiring structured data exchange with an `LLM Agent`, the ADK provides mechanisms to define expected input and desired output formats using schema definitions.

- **`input_schema` (Optional):** Define a schema representing the expected input structure. If set, the user message content passed to this agent *must* be a JSON string conforming to this schema. Your instructions should guide the user or preceding agent accordingly.
- **`output_schema` (Optional):** Define a schema representing the desired output structure. If set, the agent's final response *must* be a JSON string conforming to this schema.
- **Constraint:** Using `output_schema` enables controlled generation within the LLM but **disables the agent's ability to use tools or transfer control to other agents**. Your instructions must guide the LLM to produce JSON matching the schema directly.
- **`output_key` (Optional):** Provide a string key. If set, the text content of the agent's *final* response will be automatically saved to the session's state dictionary under this key. This is useful for passing results between agents or steps in a workflow.
- In Python, this might look like: `session.state[output_key] = agent_response_text`
- In Java: `session.state().put(outputKey, agentResponseText)`

PythonJava

The input and output schema is typically a `Pydantic` BaseModel.

```
from pydantic import BaseModel, Field

class CapitalOutput(BaseModel):
    capital: str = Field(description="The capital of the country.")

structured_capital_agent = LlmAgent(
    # ... name, model, description
    instruction="""You are a Capital Information Agent. Given a countr
    output_schema=CapitalOutput, # Enforce JSON output
    output_key="found_capital"  # Store result in state['found_capital
    # Cannot use tools=[get_capital_city] effectively here
)
```

The input and output schema is a `google.genai.types.Schema` object.

```
private static final Schema CAPITAL_OUTPUT =
    Schema.builder()
        .type("OBJECT")
        .description("Schema for capital city information.")
        .properties(
            Map.of(
                "capital",
                Schema.builder()
                    .type("STRING")
                    .description("The capital city of the country.")
                    .build()))
        .build();

LlmAgent structuredCapitalAgent =
    LlmAgent.builder()
        // ... name, model, description
        .instruction(
                "You are a Capital Information Agent. Given a country,
        .outputSchema(capitalOutput) // Enforce JSON output
```

```
        .outputKey("found_capital") // Store result in state.get("foun
        // Cannot use tools(getCapitalCity) effectively here
        .build();
```

Managing Context (`include_contents`)¶

Control whether the agent receives the prior conversation history.

- **`include_contents` (Optional, Default: `'default'`):** Determines if the `contents` (history) are sent to the LLM.
- `'default'`: The agent receives the relevant conversation history.
- `'none'`: The agent receives no prior `contents`. It operates based solely on its current instruction and any input provided in the *current* turn (useful for stateless tasks or enforcing specific contexts).

PythonJava

```
stateless_agent = LlmAgent(
    # ... other params
    include_contents='none'
)
```

```
import com.google.adk.agents.LlmAgent.IncludeContents;

LlmAgent statelessAgent =
    LlmAgent.builder()
        // ... other params
        .includeContents(IncludeContents.NONE)
        .build();
```

Planning & Code Execution¶

Currently supported in Python

For more complex reasoning involving multiple steps or executing code:

- **`planner` (Optional):** Assign a `BasePlanner` instance to enable multi-step reasoning and planning before execution. (See Multi-Agents patterns).
- **`code_executor` (Optional):** Provide a `BaseCodeExecutor` instance to allow the agent to execute code blocks (e.g., Python) found in the LLM's response. (See Tools/Built-in tools).

Putting It Together: Example¶

Code

Here's the complete basic `capital_agent`:

PythonJava

```
 # --- Full example code demonstrating LlmAgent with Tools vs. Output 
import json # Needed for pretty printing dicts

from google.adk.agents import LlmAgent
from google.adk.runners import Runner
from google.adk.sessions import InMemorySessionService
from google.genai import types
from pydantic import BaseModel, Field

# --- 1. Define Constants ---
APP_NAME = "agent_comparison_app"
USER_ID = "test_user_456"
SESSION_ID_TOOL_AGENT = "session_tool_agent_xyz"
SESSION_ID_SCHEMA_AGENT = "session_schema_agent_xyz"
MODEL_NAME = "gemini-2.0-flash"

# --- 2. Define Schemas ---

# Input schema used by both agents
class CountryInput(BaseModel):
    country: str = Field(description="The country to get information a
```

```
# Output schema ONLY for the second agent
class CapitalInfoOutput(BaseModel):
    capital: str = Field(description="The capital city of the country.
    # Note: Population is illustrative; the LLM will infer or estimate
    # as it cannot use tools when output_schema is set.
    population_estimate: str = Field(description="An estimated populat

# --- 3. Define the Tool (Only for the first agent) ---
def get_capital_city(country: str) -> str:
    """Retrieves the capital city of a given country."""
    print(f"\n-- Tool Call: get_capital_city(country='{country}') --")
    country_capitals = {
        "united states": "Washington, D.C.",
        "canada": "Ottawa",
        "france": "Paris",
        "japan": "Tokyo",
    }
    result = country_capitals.get(country.lower(), f"Sorry, I couldn't
    print(f"-- Tool Result: '{result}' --")
    return result

# --- 4. Configure Agents ---

# Agent 1: Uses a tool and output_key
capital_agent_with_tool = LlmAgent(
    model=MODEL_NAME,
    name="capital_agent_tool",
    description="Retrieves the capital city using a specific tool.",
    instruction="""You are a helpful agent that provides the capital c
The user will provide the country name in a JSON format like {"country
1. Extract the country name.
2. Use the `get_capital_city` tool to find the capital.
3. Respond clearly to the user, stating the capital city found by the
""",
    tools=[get_capital_city],
```

```
    input_schema=CountryInput,
    output_key="capital_tool_result", # Store final text response
)

# Agent 2: Uses output_schema (NO tools possible)
structured_info_agent_schema = LlmAgent(
    model=MODEL_NAME,
    name="structured_info_agent_schema",
    description="Provides capital and estimated population in a specif
    instruction=f"""You are an agent that provides country information
The user will provide the country name in a JSON format like {{"countr
Respond ONLY with a JSON object matching this exact schema:
{json.dumps(CapitalInfoOutput.model_json_schema(), indent=2)}
Use your knowledge to determine the capital and estimate the populatio
""",
    # *** NO tools parameter here - using output_schema prevents tool
    input_schema=CountryInput,
    output_schema=CapitalInfoOutput, # Enforce JSON output structure
    output_key="structured_info_result", # Store final JSON response
)

# --- 5. Set up Session Management and Runners ---
session_service = InMemorySessionService()

# Create separate sessions for clarity, though not strictly necessary
session_service.create_session(app_name=APP_NAME, user_id=USER_ID, ses
session_service.create_session(app_name=APP_NAME, user_id=USER_ID, ses

# Create a runner for EACH agent
capital_runner = Runner(
    agent=capital_agent_with_tool,
    app_name=APP_NAME,
    session_service=session_service
)
structured_runner = Runner(
    agent=structured_info_agent_schema,
```

```
    app_name=APP_NAME,
    session_service=session_service
)

# --- 6. Define Agent Interaction Logic ---
async def call_agent_and_print(
    runner_instance: Runner,
    agent_instance: LlmAgent,
    session_id: str,
    query_json: str
):
    """Sends a query to the specified agent/runner and prints results.
    print(f"\n>>> Calling Agent: '{agent_instance.name}' | Query: {que

    user_content = types.Content(role='user', parts=[types.Part(text=q

    final_response_content = "No final response received."
    async for event in runner_instance.run_async(user_id=USER_ID, sess
        # print(f"Event: {event.type}, Author: {event.author}") # Unco
        if event.is_final_response() and event.content and event.conte
            # For output_schema, the content is the JSON string itself
            final_response_content = event.content.parts[0].text

    print(f"<<< Agent '{agent_instance.name}' Response: {final_respons

    current_session = session_service.get_session(app_name=APP_NAME,
                                                  user_id=USER_ID,
                                                  session_id=session_i
    stored_output = current_session.state.get(agent_instance.output_ke

    # Pretty print if the stored output looks like JSON (likely from o
    print(f"--- Session State ['{agent_instance.output_key}']: ", end=
    try:
        # Attempt to parse and pretty print if it's JSON
        parsed_output = json.loads(stored_output)
        print(json.dumps(parsed_output, indent=2))
```

```
    except (json.JSONDecodeError, TypeError):
         # Otherwise, print as string
        print(stored_output)
    print("-" * 30)

# --- 7. Run Interactions ---
async def main():
    print("--- Testing Agent with Tool ---")
    await call_agent_and_print(capital_runner, capital_agent_with_tool
    await call_agent_and_print(capital_runner, capital_agent_with_tool

    print("\n\n--- Testing Agent with Output Schema (No Tool Use) ---"
    await call_agent_and_print(structured_runner, structured_info_agen
    await call_agent_and_print(structured_runner, structured_info_agen

if __name__ == "__main__":
    await main()
```

```
// --- Full example code demonstrating LlmAgent with Tools vs. Output

import com.google.adk.agents.LlmAgent;
import com.google.adk.events.Event;
import com.google.adk.runner.Runner;
import com.google.adk.sessions.InMemorySessionService;
import com.google.adk.sessions.Session;
import com.google.adk.tools.Annotations;
import com.google.adk.tools.FunctionTool;
import com.google.genai.types.Content;
import com.google.genai.types.Part;
import com.google.genai.types.Schema;
import io.reactivex.rxjava3.core.Flowable;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
```

```
import java.util.Optional;

public class LlmAgentExample {

  // --- 1. Define Constants ---
  private static final String MODEL_NAME = "gemini-2.0-flash";
  private static final String APP_NAME = "capital_agent_tool";
  private static final String USER_ID = "test_user_456";
  private static final String SESSION_ID_TOOL_AGENT = "session_tool_ag
  private static final String SESSION_ID_SCHEMA_AGENT = "session_schem

  // --- 2. Define Schemas ---

  // Input schema used by both agents
  private static final Schema COUNTRY_INPUT_SCHEMA =
      Schema.builder()
          .type("OBJECT")
          .description("Input for specifying a country.")
          .properties(
              Map.of(
                  "country",
                  Schema.builder()
                      .type("STRING")
                      .description("The country to get information abo
                      .build()))
          .required(List.of("country"))
          .build();

  // Output schema ONLY for the second agent
  private static final Schema CAPITAL_INFO_OUTPUT_SCHEMA =
      Schema.builder()
          .type("OBJECT")
          .description("Schema for capital city information.")
          .properties(
              Map.of(
                  "capital",
```

```
                    Schema.builder()
                        .type("STRING")
                        .description("The capital city of the country.")
                        .build(),
                    "population_estimate",
                    Schema.builder()
                        .type("STRING")
                        .description("An estimated population of the cap
                        .build()))
            .required(List.of("capital", "population_estimate"))
            .build();

    // --- 3. Define the Tool (Only for the first agent) ---
    // Retrieves the capital city of a given country.
    public static Map<String, Object> getCapitalCity(
        @Annotations.Schema(name = "country", description = "The country
        String country) {
      System.out.printf("%n-- Tool Call: getCapitalCity(country='%s') --
      Map<String, String> countryCapitals = new HashMap<>();
      countryCapitals.put("united states", "Washington, D.C.");
      countryCapitals.put("canada", "Ottawa");
      countryCapitals.put("france", "Paris");
      countryCapitals.put("japan", "Tokyo");

      String result =
          countryCapitals.getOrDefault(
              country.toLowerCase(), "Sorry, I couldn't find the capital
      System.out.printf("-- Tool Result: '%s' --%n", result);
      return Map.of("result", result); // Tools must return a Map
    }

    public static void main(String[] args){
      LlmAgentExample agentExample = new LlmAgentExample();
      FunctionTool capitalTool = FunctionTool.create(agentExample.getCla

      // --- 4. Configure Agents ---
```

```
// Agent 1: Uses a tool and output_key
LlmAgent capitalAgentWithTool =
    LlmAgent.builder()
        .model(MODEL_NAME)
        .name("capital_agent_tool")
        .description("Retrieves the capital city using a specific
        .instruction(
          """
          You are a helpful agent that provides the capital city c
          1. Extract the country name.
          2. Use the `get_capital_city` tool to find the capital.
          3. Respond clearly to the user, stating the capital city
          """)
        .tools(capitalTool)
        .inputSchema(COUNTRY_INPUT_SCHEMA)
        .outputKey("capital_tool_result") // Store final text resp
        .build();

// Agent 2: Uses an output schema
LlmAgent structuredInfoAgentSchema =
    LlmAgent.builder()
        .model(MODEL_NAME)
        .name("structured_info_agent_schema")
        .description("Provides capital and estimated population in
        .instruction(
            String.format("""
            You are an agent that provides country information.
            Respond ONLY with a JSON object matching this exact sc
            Use your knowledge to determine the capital and estima
            """, CAPITAL_INFO_OUTPUT_SCHEMA.toJson()))
        // *** NO tools parameter here - using output_schema preve
        .inputSchema(COUNTRY_INPUT_SCHEMA)
        .outputSchema(CAPITAL_INFO_OUTPUT_SCHEMA) // Enforce JSON
        .outputKey("structured_info_result") // Store final JSON r
        .build();
```

```
    // --- 5. Set up Session Management and Runners ---
    InMemorySessionService sessionService = new InMemorySessionService

    sessionService.createSession(APP_NAME, USER_ID, null, SESSION_ID_T
    sessionService.createSession(APP_NAME, USER_ID, null, SESSION_ID_S

    Runner capitalRunner = new Runner(capitalAgentWithTool, APP_NAME,
    Runner structuredRunner = new Runner(structuredInfoAgentSchema, AP

    // --- 6. Run Interactions ---
    System.out.println("--- Testing Agent with Tool ---");
    agentExample.callAgentAndPrint(
        capitalRunner, capitalAgentWithTool, SESSION_ID_TOOL_AGENT, "{
    agentExample.callAgentAndPrint(
        capitalRunner, capitalAgentWithTool, SESSION_ID_TOOL_AGENT, "{

    System.out.println("\n\n--- Testing Agent with Output Schema (No T
    agentExample.callAgentAndPrint(
        structuredRunner,
        structuredInfoAgentSchema,
        SESSION_ID_SCHEMA_AGENT,
        "{\"country\": \"France\"}");
    agentExample.callAgentAndPrint(
        structuredRunner,
        structuredInfoAgentSchema,
        SESSION_ID_SCHEMA_AGENT,
        "{\"country\": \"Japan\"}");
}

// --- 7. Define Agent Interaction Logic ---
public void callAgentAndPrint(Runner runner, LlmAgent agent, String
    System.out.printf(
        "%n>>> Calling Agent: '%s' | Session: '%s' | Query: %s%n",
        agent.name(), sessionId, queryJson);
```

```
    Content userContent = Content.fromParts(Part.fromText(queryJson));
    final String[] finalResponseContent = {"No final response received
    Flowable<Event> eventStream = runner.runAsync(USER_ID, sessionId,

    // Stream event response
    eventStream.blockingForEach(event -> {
          if (event.finalResponse() && event.content().isPresent()) {
            event
                .content()
                .get()
                .parts()
                .flatMap(parts -> parts.isEmpty() ? Optional.empty() :
                .flatMap(Part::text)
                .ifPresent(text -> finalResponseContent[0] = text);
          }
        });

    System.out.printf("<<< Agent '%s' Response: %s%n", agent.name(), f

    // Retrieve the session again to get the updated state
    Session updatedSession =
        runner
            .sessionService()
            .getSession(APP_NAME, USER_ID, sessionId, Optional.empty()
            .blockingGet();

    if (updatedSession != null && agent.outputKey().isPresent()) {
      // Print to verify if the stored output looks like JSON (likely
      System.out.printf("--- Session State ['%s']: ", agent.outputKey(
      }
  }
}
```

(This example demonstrates the core concepts. More complex agents might incorporate schemas, context control, planning, etc.)

Related Concepts (Deferred Topics)¶

While this page covers the core configuration of `LlmAgent`, several related concepts provide more advanced control and are detailed elsewhere:

- **Callbacks:** Intercepting execution points (before/after model calls, before/after tool calls) using `before_model_callback`, `after_model_callback`, etc. See Callbacks.
- **Multi-Agent Control:** Advanced strategies for agent interaction, including planning (`planner`), controlling agent transfer (`disallow_transfer_to_parent`, `disallow_transfer_to_peers`), and system-wide instructions (`global_instruction`). See Multi-Agents.