Models - Agent Development Kit

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

Using Different Models with ADK¶

Note

Java ADK currently supports Gemini and Anthropic models. More model support coming soon.

The Agent Development Kit (ADK) is designed for flexibility, allowing you to integrate various Large Language Models (LLMs) into your agents. While the setup for Google Gemini models is covered in the Setup Foundation Models guide, this page details how to leverage Gemini effectively and integrate other popular models, including those hosted externally or running locally.

ADK primarily uses two mechanisms for model integration:

1. **Direct String / Registry:** For models tightly integrated with Google Cloud (like Gemini models accessed via Google AI Studio or Vertex AI) or models hosted on Vertex AI endpoints. You typically provide the model name or endpoint resource string directly to the `LlmAgent`. ADK's internal registry resolves this string to the appropriate backend client, often utilizing the `google-genai` library.
2. **Wrapper Classes:** For broader compatibility, especially with models outside the Google ecosystem or those requiring specific client configurations (like models accessed via LiteLLM). You instantiate a specific wrapper class (e.g., `LiteLlm`) and pass this object as the `model` parameter to your `LlmAgent`.

The following sections guide you through using these methods based on your needs.

Using Google Gemini Models¶

This is the most direct way to use Google's flagship models within ADK.

Integration Method: Pass the model's identifier string directly to the `model` parameter of `LlmAgent` (or its alias, `Agent`).

Backend Options & Setup:

The `google-genai` library, used internally by ADK for Gemini, can connect through either Google AI Studio or Vertex AI.

Model support for voice/video streaming

In order to use voice/video streaming in ADK, you will need to use Gemini models that support the Live API. You can find the **model ID(s)** that support the Gemini Live API in the documentation:

- Google AI Studio: Gemini Live API
- Vertex AI: Gemini Live API

Google AI Studio¶

- **Use Case:** Google AI Studio is the easiest way to get started with Gemini. All you need is the API key. Best for rapid prototyping and development.
- **Setup:** Typically requires an API key:
- Set as an environment variable or
- Passed during the model initialization via the `Client` (see example below)

```
export GOOGLE_API_KEY="YOUR_GOOGLE_API_KEY"
export GOOGLE_GENAI_USE_VERTEXAI=FALSE
```

- **Models:** Find all available models on the Google AI for Developers site.

Vertex AI¶

- **Use Case:** Recommended for production applications, leveraging Google Cloud infrastructure. Gemini on Vertex AI supports enterprise-grade features, security, and compliance controls.

- **Setup:**

• Authenticate using Application Default Credentials (ADC):

``` gcloud auth application-default login

``` + Configure these variables either as environment variables or by providing them directly when initializing the Model.

Set your Google Cloud project and location:

``` export GOOGLE_CLOUD_PROJECT="YOUR_PROJECT_ID" export GOOGLE_CLOUD_LOCATION="YOUR_VERTEX_AI_LOCATION" # e.g., us-central1

```

Explicitly tell the library to use Vertex AI:

``` export GOOGLE_GENAI_USE_VERTEXAI=TRUE

``` * **Models:** Find available model IDs in the Vertex AI documentation.

Example:

PythonJava

```
from google.adk.agents import LlmAgent

# --- Example using a stable Gemini Flash model ---
agent_gemini_flash = LlmAgent(
    # Use the latest stable Flash model identifier
    model="gemini-2.0-flash",
    name="gemini_flash_agent",
    instruction="You are a fast and helpful Gemini assistant.",
    # ... other agent parameters
)

# --- Example using a powerful Gemini Pro model ---
# Note: Always check the official Gemini documentation for the latest
# including specific preview versions if needed. Preview models might
# different availability or quota limitations.
```

```
agent_gemini_pro = LlmAgent(
    # Use the latest generally available Pro model identifier
    model="gemini-2.5-pro-preview-03-25",
    name="gemini_pro_agent",
    instruction="You are a powerful and knowledgeable Gemini assistant
    # ... other agent parameters
)
```

```
// --- Example #1: using a stable Gemini Flash model with ENV variable
LlmAgent agentGeminiFlash =
    LlmAgent.builder()
        // Use the latest stable Flash model identifier
        .model("gemini-2.0-flash") // Set ENV variables to use this mo
        .name("gemini_flash_agent")
        .instruction("You are a fast and helpful Gemini assistant.")
        // ... other agent parameters
        .build();

// --- Example #2: using a powerful Gemini Pro model with API Key in m
LlmAgent agentGeminiPro =
    LlmAgent.builder()
        // Use the latest generally available Pro model identifier
        .model(new Gemini("gemini-2.5-pro-preview-03-25",
            Client.builder()
                .vertexAI(false)
                .apiKey("API_KEY") // Set the API Key (or) project/ lo
                .build()))
        // Or, you can also directly pass the API_KEY
        // .model(new Gemini("gemini-2.5-pro-preview-03-25", "API_KEY"
        .name("gemini_pro_agent")
        .instruction("You are a powerful and knowledgeable Gemini assi
        // ... other agent parameters
        .build();
```

```
// Note: Always check the official Gemini documentation for the latest
// including specific preview versions if needed. Preview models might
// different availability or quota limitations.
```

Using Anthropic models¶

Supported in Java

You can integrate Anthropic's Claude models directly using their API key or from a Vertex AI backend into your Java ADK applications by using the ADK's `Claude` wrapper class.

For Vertex AI backend, see the Third-Party Models on Vertex AI section.

Prerequisites:

1. **Dependencies:**
2. **Anthropic SDK Classes (Transitive):** The Java ADK's `com.google.adk.models.Claude` wrapper relies on classes from Anthropic's official Java SDK. These are typically included as **transitive dependencies**.
3. **Anthropic API Key:**
4. Obtain an API key from Anthropic. Securely manage this key using a secret manager.

Integration:

Instantiate `com.google.adk.models.Claude`, providing the desired Claude model name and an `AnthropicOkHttpClient` configured with your API key. Then, pass this `Claude` instance to your `LlmAgent`.

Example:

```
import com.anthropic.client.AnthropicClient;
import com.google.adk.agents.LlmAgent;
import com.google.adk.models.Claude;
```

```
import com.anthropic.client.okhttp.AnthropicOkHttpClient; // From Anth

public class DirectAnthropicAgent {

  private static final String CLAUDE_MODEL_ID = "claude-3-7-sonnet-lat

  public static LlmAgent createAgent() {

    // It's recommended to load sensitive keys from a secure config
    AnthropicClient anthropicClient = AnthropicOkHttpClient.builder()
        .apiKey("ANTHROPIC_API_KEY")
        .build();

    Claude claudeModel = new Claude(
        CLAUDE_MODEL_ID,
        anthropicClient
    );

    return LlmAgent.builder()
        .name("claude_direct_agent")
        .model(claudeModel)
        .instruction("You are a helpful AI assistant powered by Anthro
        // ... other LlmAgent configurations
        .build();
  }

  public static void main(String[] args) {
    try {
      LlmAgent agent = createAgent();
      System.out.println("Successfully created direct Anthropic agent:
    } catch (IllegalStateException e) {
      System.err.println("Error creating agent: " + e.getMessage());
    }
  }
}
```

Using Cloud & Proprietary Models via LiteLLM¶

Supported in Python

To access a vast range of LLMs from providers like OpenAI, Anthropic (non-Vertex AI), Cohere, and many others, ADK offers integration through the LiteLLM library.

Integration Method: Instantiate the `LiteLlm` wrapper class and pass it to the `model` parameter of `LlmAgent`.

LiteLLM Overview: LiteLLM acts as a translation layer, providing a standardized, OpenAI-compatible interface to over 100+ LLMs.

Setup:

1. **Install LiteLLM:**

``` pip install litellm

``` 2. **Set Provider API Keys:** Configure API keys as environment variables for the specific providers you intend to use.

• *Example for OpenAI:*

``` export OPENAI_API_KEY="YOUR_OPENAI_API_KEY"

``` * *Example for Anthropic (non-Vertex AI):*

``` export ANTHROPIC_API_KEY="YOUR_ANTHROPIC_API_KEY"

``` * *Consult the LiteLLM Providers Documentation for the correct environment variable names for other providers.*

Example:

``` from google.adk.agents import LlmAgent from google.adk.models.lite_llm import LiteLlm

# --- Example Agent using OpenAI's GPT-4o --- # (Requires OPENAI_API_KEY) agent_openai = LlmAgent( model=LiteLlm(model="openai/gpt-4o"), # LiteLLM model
```

string format name="openai_agent", instruction="You are a helpful assistant powered by GPT-4o.", # ... other agent parameters)

--- Example Agent using Anthropic's Claude Haiku (non-Vertex) --- # (Requires ANTHROPIC_API_KEY) agent_claude_direct = LlmAgent(model=LiteLlm(model="anthropic/claude-3-haiku-20240307"), name="claude_direct_agent", instruction="You are an assistant powered by Claude Haiku.", # ... other agent parameters)

```

Note for Windows users

### Avoiding LiteLLM UnicodeDecodeError on Windows¶

When using ADK agents with LiteLlm on Windows, users might encounter the following error:

```
UnicodeDecodeError: 'charmap' codec can't decode byte...
```

This issue occurs because `litellm` (used by LiteLlm) reads cached files (e.g., model pricing information) using the default Windows encoding (`cp1252`) instead of UTF-8. Windows users can prevent this issue by setting the `PYTHONUTF8` environment variable to `1`. This forces Python to use UTF-8 globally. **Example (PowerShell):**

```
 # Set for current session
$env:PYTHONUTF8 = "1"
# Set persistently for the user
[System.Environment]::SetEnvironmentVariable('PYTHONUTF8', '1', [Syste
Applying this setting ensures that Python reads cached files using UTF
```

Using Open & Local Models via LiteLLM¶

Supported in Python
```

For maximum control, cost savings, privacy, or offline use cases, you can run open-source models locally or self-host them and integrate them using LiteLLM.

Integration Method: Instantiate the `LiteLlm` wrapper class, configured to point to your local model server.

Ollama Integration¶

Ollama allows you to easily run open-source models locally.

Model choice¶

If your agent is relying on tools, please make sure that you select a model with tool support from Ollama website.

For reliable results, we recommend using a decent-sized model with tool support.

The tool support for the model can be checked with the following command:

```
ollama show mistral-small3.1
  Model
    architecture        mistral3
    parameters          24.0B
    context length      131072
    embedding length    5120
    quantization        Q4_K_M

  Capabilities
    completion
    vision
    tools
```

You are supposed to see `tools` listed under capabilities.

You can also look at the template the model is using and tweak it based on your needs.

```
ollama show --modelfile llama3.2 > model_file_to_modify
```

For instance, the default template for the above model inherently suggests that the model shall call a function all the time. This may result in an infinite loop of function calls.

```
Given the following functions, please respond with a JSON for a functi
with its proper arguments that best answers the given prompt.

Respond in the format {"name": function name, "parameters": dictionary
argument name and its value}. Do not use variables.
```

You can swap such prompts with a more descriptive one to prevent infinite tool call loops.

For instance:

```
Review the user's prompt and the available functions listed below.
First, determine if calling one of these functions is the most appropr

If you determine a function call IS required: Respond ONLY with a JSON

If you determine a function call IS NOT required: Respond directly to
```

Then you can create a new model with the following command:

```
ollama create llama3.2-modified -f model_file_to_modify
```

Using ollama_chat provider¶

Our LiteLLM wrapper can be used to create agents with Ollama models.

```
root_agent = Agent(
    model=LiteLlm(model="ollama_chat/mistral-small3.1"),
    name="dice_agent",
    description=(
        "hello world agent that can roll a dice of 8 sides and check p
        " numbers."
    ),
    instruction="""
      You roll dice and answer questions about the outcome of the dice
    """,
    tools=[
        roll_die,
        check_prime,
    ],
)
```

It is important to set the provider `ollama_chat` instead of `ollama`. Using `ollama` will result in unexpected behaviors such as infinite tool call loops and ignoring previous context.

While `api_base` can be provided inside LiteLLM for generation, LiteLLM library is calling other APIs relying on the env variable instead as of v1.65.5 after completion. So at this time, we recommend setting the env variable `OLLAMA_API_BASE` to point to the ollama server.

```
export OLLAMA_API_BASE="http://localhost:11434"
adk web
```

Using openai provider¶

Alternatively, `openai` can be used as the provider name. But this will also require setting the `OPENAI_API_BASE=http://localhost:11434/v1` and `OPENAI_API_KEY=anything` env variables instead of `OLLAMA_API_BASE`. **Please note that api base now has `/v1` at the end.**

```
root_agent = Agent(
    model=LiteLlm(model="openai/mistral-small3.1"),
    name="dice_agent",
    description=(
        "hello world agent that can roll a dice of 8 sides and check p
        " numbers."
    ),
    instruction="""
      You roll dice and answer questions about the outcome of the dice
    """,
    tools=[
        roll_die,
        check_prime,
    ],
)
```

```
export OPENAI_API_BASE=http://localhost:11434/v1
export OPENAI_API_KEY=anything
adk web
```

Debugging¶

You can see the request sent to the Ollama server by adding the following in your agent code just after imports.

```
import litellm
litellm._turn_on_debug()
```

Look for a line like the following:

```
Request Sent from LiteLLM:
curl -X POST \
http://localhost:11434/api/chat \
```

```
-d '{'model': 'mistral-small3.1', 'messages': [{'role': 'system', 'con
```

Self-Hosted Endpoint (e.g., vLLM)¶

Supported in Python

Tools such as vLLM allow you to host models efficiently and often expose an OpenAI-compatible API endpoint.

Setup:

1. **Deploy Model:** Deploy your chosen model using vLLM (or a similar tool). Note the API base URL (e.g., `https://your-vllm-endpoint.run.app/v1`).
2. *Important for ADK Tools:* When deploying, ensure the serving tool supports and enables OpenAI-compatible tool/function calling. For vLLM, this might involve flags like `--enable-auto-tool-choice` and potentially a specific `--tool-call-parser`, depending on the model. Refer to the vLLM documentation on Tool Use.
3. **Authentication:** Determine how your endpoint handles authentication (e.g., API key, bearer token).

Integration Example:

``` import subprocess from google.adk.agents import LlmAgent from google.adk.models.lite_llm import LiteLlm

# --- Example Agent using a model hosted on a vLLM endpoint ---

# Endpoint URL provided by your vLLM deployment api_base_url = "https://your-vllm-endpoint.run.app/v1"

# Model name as recognized by *your* vLLM endpoint configuration model_name_at_endpoint = "hosted_vllm/google/gemma-3-4b-it" # Example from vllm_test.py

# Authentication (Example: using gcloud identity token for a Cloud Run deployment) # Adapt this based on your endpoint's security try: gcloud_token = subprocess.check_output( ["gcloud", "auth", "print-identity-token", "-q"] ).decode().strip() auth_headers = {"Authorization": f"Bearer {gcloud_token}"}
```

except Exception as e: print(f"Warning: Could not get gcloud token - {e}. Endpoint might be unsecured or require different auth.") auth_headers = None # Or handle error appropriately

agent_vllm = LlmAgent(model=LiteLlm(model=model_name_at_endpoint, api_base=api_base_url, # Pass authentication headers if needed extra_headers=auth_headers # Alternatively, if endpoint uses an API key: # api_key="YOUR_ENDPOINT_API_KEY"), name="vllm_agent", instruction="You are a helpful assistant running on a self-hosted vLLM endpoint.", # ... other agent parameters)

```

## Using Hosted & Tuned Models on Vertex AI¶

For enterprise-grade scalability, reliability, and integration with Google Cloud's MLOps ecosystem, you can use models deployed to Vertex AI Endpoints. This includes models from Model Garden or your own fine-tuned models.

**Integration Method:** Pass the full Vertex AI Endpoint resource string (`projects/PROJECT_ID/locations/LOCATION/endpoints/ENDPOINT_ID`) directly to the `model` parameter of `LlmAgent`.

**Vertex AI Setup (Consolidated):**

Ensure your environment is configured for Vertex AI:

1. **Authentication:** Use Application Default Credentials (ADC):

``` gcloud auth application-default login

``` 2. **Environment Variables:** Set your project and location:

``` export GOOGLE_CLOUD_PROJECT="YOUR_PROJECT_ID" export GOOGLE_CLOUD_LOCATION="YOUR_VERTEX_AI_LOCATION" # e.g., us-central1

`` 3. **Enable Vertex Backend:** Crucially, ensure the google-genai` library targets Vertex AI:

``` export GOOGLE_GENAI_USE_VERTEXAI=TRUE
```

```

## Model Garden Deployments¶

Currently supported in Python

You can deploy various open and proprietary models from the Vertex AI Model Garden to an endpoint.

**Example:**

```
from google.adk.agents import LlmAgent
from google.genai import types # For config objects

--- Example Agent using a Llama 3 model deployed from Model Garden -

# Replace with your actual Vertex AI Endpoint resource name
llama3_endpoint = "projects/YOUR_PROJECT_ID/locations/us-central1/endp

agent_llama3_vertex = LlmAgent(
    model=llama3_endpoint,
    name="llama3_vertex_agent",
    instruction="You are a helpful assistant based on Llama 3, hosted
    generate_content_config=types.GenerateContentConfig(max_output_tok
    # ... other agent parameters
)
```

## Fine-tuned Model Endpoints¶

Currently supported in Python

Deploying your fine-tuned models (whether based on Gemini or other architectures supported by Vertex AI) results in an endpoint that can be used directly.

**Example:**
```

```
from google.adk.agents import LlmAgent

# --- Example Agent using a fine-tuned Gemini model endpoint ---

# Replace with your fine-tuned model's endpoint resource name
finetuned_gemini_endpoint = "projects/YOUR_PROJECT_ID/locations/us-cen

agent_finetuned_gemini = LlmAgent(
    model=finetuned_gemini_endpoint,
    name="finetuned_gemini_agent",
    instruction="You are a specialized assistant trained on specific d
    # ... other agent parameters
)
```

Third-Party Models on Vertex AI (e.g., Anthropic Claude)¶

Some providers, like Anthropic, make their models available directly through Vertex AI.

PythonJava

Integration Method: Uses the direct model string (e.g., `"claude-3-sonnet@20240229"`), *but requires manual registration* within ADK.

Why Registration? ADK's registry automatically recognizes `gemini-*` strings and standard Vertex AI endpoint strings (`projects/.../endpoints/...`) and routes them via the `google-genai` library. For other model types used directly via Vertex AI (like Claude), you must explicitly tell the ADK registry which specific wrapper class (`Claude` in this case) knows how to handle that model identifier string with the Vertex AI backend.

Setup:

1. **Vertex AI Environment:** Ensure the consolidated Vertex AI setup (ADC, Env Vars, `GOOGLE_GENAI_USE_VERTEXAI=TRUE`) is complete.
2. **Install Provider Library:** Install the necessary client library configured for Vertex AI.

``` pip install "anthropic[vertex]"

``` 3. **Register Model Class:** Add this code near the start of your application, *before* creating an agent using the Claude model string:

``` # Required for using Claude model strings directly via Vertex AI with LlmAgent from google.adk.models.anthropic_llm import Claude from google.adk.models.registry import LLMRegistry

LLMRegistry.register(Claude)

```

**Example:**

```
 from google.adk.agents import LlmAgent
from google.adk.models.anthropic_llm import Claude # Import needed for
from google.adk.models.registry import LLMRegistry # Import needed for
from google.genai import types

# --- Register Claude class (do this once at startup) ---
LLMRegistry.register(Claude)

--- Example Agent using Claude 3 Sonnet on Vertex AI ---

# Standard model name for Claude 3 Sonnet on Vertex AI
claude_model_vertexai = "claude-3-sonnet@20240229"

agent_claude_vertexai = LlmAgent(
    model=claude_model_vertexai, # Pass the direct string after regist
    name="claude_vertexai_agent",
    instruction="You are an assistant powered by Claude 3 Sonnet on Ve
    generate_content_config=types.GenerateContentConfig(max_output_tok
    # ... other agent parameters
)
```

Integration Method: Directly instantiate the provider-specific model class (e.g., `com.google.adk.models.Claude`) and configure it with a Vertex AI backend.

Why Direct Instantiation? The Java ADK's `LlmRegistry` primarily handles Gemini models by default. For third-party models like Claude on Vertex AI, you directly provide an instance of the ADK's wrapper class (e.g., `Claude`) to the `LlmAgent`. This wrapper class is responsible for interacting with the model via its specific client library, configured for Vertex AI.

Setup:

1. **Vertex AI Environment:**
2. Ensure your Google Cloud project and region are correctly set up.
3. **Application Default Credentials (ADC):** Make sure ADC is configured correctly in your environment. This is typically done by running `gcloud auth application-default login`. The Java client libraries will use these credentials to authenticate with Vertex AI. Follow the Google Cloud Java documentation on ADC for detailed setup.
4. **Provider Library Dependencies:**
5. **Third-Party Client Libraries (Often Transitive):** The ADK core library often includes the necessary client libraries for common third-party models on Vertex AI (like Anthropic's required classes) as **transitive dependencies**. This means you might not need to explicitly add a separate dependency for the Anthropic Vertex SDK in your `pom.xml` or `build.gradle`.
6. **Instantiate and Configure the Model:** When creating your `LlmAgent`, instantiate the `Claude` class (or the equivalent for another provider) and configure its `VertexBackend`.

Example:

```
import com.anthropic.client.AnthropicClient;
import com.anthropic.client.okhttp.AnthropicOkHttpClient;
import com.anthropic.vertex.backends.VertexBackend;
```

```
import com.google.adk.agents.LlmAgent;
import com.google.adk.models.Claude; // ADK's wrapper for Claude
import com.google.auth.oauth2.GoogleCredentials;
import java.io.IOException;

// ... other imports

public class ClaudeVertexAiAgent {

    public static LlmAgent createAgent() throws IOException {
        // Model name for Claude 3 Sonnet on Vertex AI (or other versi
        String claudeModelVertexAi = "claude-3-7-sonnet"; // Or any ot

        // Configure the AnthropicOkHttpClient with the VertexBackend
        AnthropicClient anthropicClient = AnthropicOkHttpClient.builde
            .backend(
                VertexBackend.builder()
                    .region("us-east5") // Specify your Vertex AI regi
                    .project("your-gcp-project-id") // Specify your GC
                    .googleCredentials(GoogleCredentials.getApplicatio
                    .build())
            .build();

        // Instantiate LlmAgent with the ADK Claude wrapper
        LlmAgent agentClaudeVertexAi = LlmAgent.builder()
            .model(new Claude(claudeModelVertexAi, anthropicClient)) /
            .name("claude_vertexai_agent")
            .instruction("You are an assistant powered by Claude 3 Son
            // .generateContentConfig(...) // Optional: Add generation
            // ... other agent parameters
            .build();

        return agentClaudeVertexAi;
    }

    public static void main(String[] args) {
```

```
        try {
            LlmAgent agent = createAgent();
            System.out.println("Successfully created agent: " + agent.
            // Here you would typically set up a Runner and Session to
        } catch (IOException e) {
            System.err.println("Failed to create agent: " + e.getMessa
            e.printStackTrace();
        }
    }
}
```