

# Tutorial 28: Using Other LLMs - Multi-Model Support

---

**Difficulty:** advanced

**Reading Time:** 1.5 hours

**Tags:** advanced, llms, multi-model, providers, configuration

**Description:** Configure agents to work with different LLM providers including OpenAI, Anthropic, and local models for diverse AI capabilities.

## Tutorial 28: Using Other LLMs with LiteLLM

---

**Goal:** Use OpenAI, Claude, Ollama, and other LLMs in your ADK agents via LiteLLM

### Prerequisites:

- Tutorial 01 (Hello World Agent)
- Tutorial 22 (Model Selection & Configuration)
- Basic understanding of API keys and environment variables

### What You'll Learn:

- Using OpenAI models (GPT-4o-mini) with ADK
- Using Anthropic Claude models (3.7 Sonnet) with ADK
- Running local models with Ollama (Granite 4) for privacy
- Multi-provider comparison and cost optimization
- When NOT to use LiteLLM
- Best practices for cross-provider development

---

**Source:** `google/adk/models/lite_llm.py`,  
`contributing/samples/hello_world_litellm/`,  
`contributing/samples/hello_world_ollama/`

---

## Why Use LiteLLM?

---

**LiteLLM** enables ADK agents to use **100+ LLM providers** with a unified interface.

### When to Use LiteLLM:

- ✓ Need OpenAI models (GPT-4o, GPT-4o-mini)
- ✓ Want Anthropic Claude (3.7 Sonnet, Opus, Haiku)
- ✓ Running local models with Ollama (privacy, cost, offline)
- ✓ Azure OpenAI (enterprise contracts)
- ✓ Multi-provider strategy (fallback, cost optimization)
- ✓ Comparing model performance across providers

### When NOT to Use LiteLLM:

- ✗ **Using Gemini models** → Use native `GoogleGenAI` (better performance, features)
  - ✗ Simple prototype with just Gemini
  - ✗ When you need Gemini-specific features (`thinking_config`, grounding)
- 

## 1. OpenAI Integration

---

**OpenAI's GPT models** are widely used for their strong reasoning and instruction-following.

### | Setup

#### 1. Install dependencies:

```
pip install google-adk[litellm]  
# Or manually:  
pip install litellm openai
```

**2. Get API key** from [OpenAI Platform](https://platform.openai.com/api-keys) (<https://platform.openai.com/api-keys>)

**3. Set environment variable:**

## | Example: GPT-4o Agent

```

"""
ADK agent using OpenAI GPT-4o via LiteLLM.
Source: contributing/samples/hello_world_litellm/agent.py
"""

import asyncio
import os
from google.adk.agents import Agent
from google.adk.runners import InMemoryRunner
from google.adk.models import LiteLlm
from google.adk.tools import FunctionTool
from google.genai import types

# Environment setup
os.environ['OPENAI_API_KEY'] = 'sk-...' # Your OpenAI API key

def calculate_square(number: int) -> int:
    """Calculate the square of a number."""
    return number ** 2

async def main():
    """Agent using OpenAI GPT-4o."""

    # Create LiteLLM model - format: "openai/model-name"
    gpt4o_model = LiteLlm(model='openai/gpt-4o-mini') # or 'openai/gpt-4o'

    # Create agent with OpenAI model
    agent = Agent(
        model=gpt4o_model, # Use LiteLlm instance, not string
        name='gpt4o_agent',
        description='Agent powered by OpenAI GPT-4o',
        instruction='You are a helpful assistant.',
        tools=[FunctionTool(calculate_square)]
    )

    # Create runner and session
    runner = InMemoryRunner(agent=agent, app_name='gpt4o_app')
    session = await runner.session_service.create_session(
        app_name='gpt4o_app',
        user_id='user_001'
    )

    # Run query with async iteration
    query = "What is the square of 12?"
    new_message = types.Content(
        role='user',
        parts=[types.Part(text=query)]
    )

```

```

    )

    async for event in runner.run_async(
        user_id='user_001',
        session_id=session.id,
        new_message=new_message
    ):
        if event.content and event.content.parts:
            print(event.content.parts[0].text)

if __name__ == '__main__':
    asyncio.run(main())

```

**Output:**

The square of 12 is 144.

## | GPT-4o-mini (Cost-Optimized)

**GPT-4o-mini** is **60x cheaper** than GPT-4o for simple tasks.

```

from google.adk.models import LiteLlm

# GPT-4o: $2.50/1M input tokens, $10/1M output tokens
gpt4o = LiteLlm(model='openai/gpt-4o')

# GPT-4o-mini: $0.15/1M input tokens, $0.60/1M output tokens
gpt4o_mini = LiteLlm(model='openai/gpt-4o-mini')

# Use mini for routine tasks
routine_agent = Agent(
    model=gpt4o_mini,
    instruction='You handle simple queries quickly.'
)

# Use full GPT-4o for complex reasoning
complex_agent = Agent(
    model=gpt4o,
    instruction='You solve complex multi-step problems.'
)

```

## Available OpenAI Models

Model	Input Cost	Output Cost	Best For
<code>openai/gpt-4o</code>	\$2.50/1M tokens	\$10/1M tokens	Complex reasoning, coding
<code>openai/gpt-4o-mini</code>	\$0.15/1M tokens	\$0.60/1M tokens	Simple tasks, high volume
<code>openai/o1</code>	\$15/1M tokens	\$60/1M tokens	Advanced reasoning chains
<code>openai/o1-mini</code>	\$3/1M tokens	\$12/1M tokens	STEM reasoning

**Model string format:** `openai/[model-name]`

## 2. Anthropic Claude Integration

**Anthropic's Claude** excels at long-form content, analysis, and following complex instructions.

### Claude Setup

**1. Install dependencies:**

```
pip install google-adk[litellm] anthropic
```

**2. Get API key** from [Anthropic Console](https://console.anthropic.com/) (<https://console.anthropic.com/>)

**3. Set environment variable:**

## | Example: Claude 3.7 Sonnet Agent



```

"""
ADK agent using Anthropic Claude 3.7 Sonnet via LiteLLM.
"""

import asyncio
import os
from google.adk.agents import Agent
from google.adk.runners import InMemoryRunner
from google.adk.models import LiteLlm
from google.adk.tools import FunctionTool
from google.genai import types

# Environment setup
os.environ['ANTHROPIC_API_KEY'] = 'sk-ant-...' # Your Anthropic API key

def analyze_sentiment(text: str) -> dict:
    """Analyze sentiment of text (mock implementation)."""
    # In production, use actual sentiment analysis
    return {
        'sentiment': 'positive',
        'confidence': 0.85,
        'key_phrases': ['exciting', 'innovative', 'breakthrough']
    }

async def main():
    """Agent using Claude 3.7 Sonnet."""

    # Create LiteLLM model - format: "anthropic/model-name"
    claude_model = LiteLlm(model='anthropic/claude-3-7-sonnet-20250219')

    # Create agent with Claude model
    agent = Agent(
        model=claude_model,
        name='claude_agent',
        description='Agent powered by Claude 3.7 Sonnet',
        instruction="""
You are a thoughtful analyst who provides detailed, nuanced responses.
You excel at:
- Complex reasoning
- Long-form content
- Ethical considerations
- Following detailed instructions
        """.strip(),
        tools=[FunctionTool(analyze_sentiment)]
    )

    # Create runner and session

```

```
runner = InMemoryRunner(agent=agent, app_name='claude_app')
session = await runner.session_service.create_session(
    app_name='claude_app',
    user_id='user_001'
)

query = """
Analyze the sentiment of this product review and explain your reasoning:
"This new AI assistant is absolutely brilliant! It understands context
incredibly well and provides helpful, accurate responses. The interface
is intuitive and the speed is impressive. Highly recommended!"
""".strip()

# Run query with async iteration
new_message = types.Content(
    role='user',
    parts=[types.Part(text=query)]
)

async for event in runner.run_async(
    user_id='user_001',
    session_id=session.id,
    new_message=new_message
):
    if event.content and event.content.parts:
        print(event.content.parts[0].text)

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

**Output:**

I'll analyze this product review's sentiment:

**Overall Sentiment**: Strongly Positive

**Analysis**:

The review exhibits overwhelmingly positive sentiment through several indicators:

- Superlative Language**: "absolutely brilliant", "incredibly well", "Highly recommended" - these are emphatic positive descriptors
- Specific Praise**: The reviewer highlights multiple strengths:
  - Contextual understanding
  - Helpful and accurate responses
  - Intuitive interface
  - Impressive speed
- Exclamation Points**: Two instances (!!) signal enthusiasm
- Recommendation**: Explicit endorsement ("Highly recommended") shows strong satisfaction
- No Criticisms**: Complete absence of negative comments or caveats

**Confidence**: 95% - The language is unambiguous and consistently positive throughout.

**Key Emotional Tone**: Enthusiastic appreciation and satisfaction

## Available Claude Models

Model	Input Cost	Output Cost	Context	Best For
anthropic/claude-3-7-sonnet-20250219	\$3/1M tokens	\$15/1M tokens	200K	Balanced (most popular)
anthropic/claude-3-5-opus-20240229	\$15/1M tokens	\$75/1M tokens	200K	Complex reasoning
anthropic/claude-3-5-haiku-20241022	\$0.80/1M tokens	\$4/1M tokens	200K	Fast, simple tasks

---

**Model string format:** `anthropic/[model-name-with-date]`

**Note:** Claude 3.7 Sonnet is the **default recommended model** (as of Q1 2025).

---

## 3. Ollama Local Models

---

**Ollama** lets you run LLMs **locally** for privacy, cost savings, and offline operation.

### | Why Use Ollama?

#### Benefits:

- ✓ **Privacy:** Data never leaves your machine
- ✓ **Cost:** No API costs after initial download
- ✓ **Offline:** Works without internet
- ✓ **Compliance:** Keep sensitive data on-premises
- ✓ **Experimentation:** Try many models freely

#### Trade-offs:

- ✗ Requires GPU for good performance
- ✗ Quality lower than GPT-4o/Claude/Gemini for complex tasks
- ✗ Slower inference on CPU
- ✗ Limited context window (typically 4K-32K vs. 200K for cloud models)

### | Ollama Setup

#### 1. Install Ollama:

```
# macOS
brew install ollama

# Linux
curl -fsSL https://ollama.com/install.sh | sh

# Windows
# Download from https://ollama.com/download
```

## 2. Start Ollama server:

```
ollama serve
# Runs on http://localhost:11434 by default
```

## 3. Pull a model:

```
# Granite 4 (IBM, strong reasoning, 8B parameters)
ollama pull granite4:latest

# Llama 3.3 (Meta, high quality, 70B parameters)
ollama pull llama3.3

# Mistral (7B parameters, fast)
ollama pull mistral

# Phi-4 (14B parameters, Microsoft, good coding)
ollama pull phi4
```

## 4. Install Python dependencies:

```
pip install google-adk[litellm]
```

**⚠ CRITICAL: Use** `ollama_chat`, **NOT** `ollama`

**WRONG ❌:**

```
# This WON'T work correctly!
model = LiteLlm(model='ollama/llama3.3') # ❌ WRONG
```

**CORRECT** ✓:

```
# Always use ollama_chat prefix!
model = LiteLlm(model='ollama_chat/llama3.3') # ✓ CORRECT
```

**Why?** LiteLLM has two Ollama interfaces:

- `ollama/` - Uses completion API (legacy, limited)
- `ollama_chat/` - Uses chat API (recommended, full features)

ADK agents require the **chat API** for proper function calling and multi-turn conversations.

## | Example: Granite 4 Local Agent

```

"""
ADK agent using local Granite 4 via Ollama.
Source: tutorial_implementation/tutorial28/multi_llm_agent/agent.py
"""

import asyncio
import os
from google.adk.agents import Agent
from google.adk.runners import InMemoryRunner
from google.adk.models import LiteLlm
from google.adk.tools import FunctionTool
from google.genai import types

# Environment setup for Ollama
os.environ['OLLAMA_API_BASE'] = 'http://localhost:11434'

def get_weather(city: str) -> dict:
    """Get current weather for a city (mock)."""
    # In production, call real weather API
    return {
        'city': city,
        'temperature': 72,
        'condition': 'Sunny',
        'humidity': 45
    }

async def main():
    """Agent using local Granite 4 model."""

    # Create LiteLLM model - format: "ollama_chat/model-name"
    # ⚠️ IMPORTANT: Use ollama_chat, NOT ollama!
    granite_model = LiteLlm(model='ollama_chat/granite4:latest')

    # Create agent with local model
    agent = Agent(
        model=granite_model,
        name='local_agent',
        description='Agent running locally with Granite 4',
        instruction='You are a helpful local assistant powered by IBM Granite'
        tools=[FunctionTool(get_weather)]
    )

    # Create runner and session
    runner = InMemoryRunner(agent=agent, app_name='ollama_app')
    session = await runner.session_service.create_session(
        app_name='ollama_app',
        user_id='user_001'
    )

```



```

)

print("\n" + "="*60)
print("LOCAL OLLAMA AGENT (Privacy-First)")
print("="*60 + "\n")

# Run query with async iteration
query = "What's the weather like in San Francisco?"
new_message = types.Content(
    role='user',
    parts=[types.Part(text=query)]
)

async for event in runner.run_async(
    user_id='user_001',
    session_id=session.id,
    new_message=new_message
):
    if event.content and event.content.parts:
        print(event.content.parts[0].text)

print("\n" + "="*60 + "\n")

if __name__ == '__main__':
    asyncio.run(main())

```

**Output:**

```

=====
LOCAL OLLAMA AGENT (Privacy-First)
=====

The weather in San Francisco is currently sunny with a temperature
of 72°F and 45% humidity. It's a beautiful day!

[All processing done locally - no data sent to cloud]

=====

```

**Output:**

```
=====
LOCAL OLLAMA AGENT (Privacy-First)
=====
```

```
The weather in San Francisco is currently sunny with a temperature
of 72°F and 45% humidity. It's a beautiful day!
```

```
[All processing done locally - no data sent to cloud]
```

## Popular Ollama Models

Model	Size	Best For	GPU RAM
<code>ollama_chat/granite4:latest</code>	8B	IBM Granite, strong reasoning	12GB
<code>ollama_chat/llama3.3</code>	70B	General tasks, strong reasoning	40GB+
<code>ollama_chat/llama3.2</code>	3B	Fast, low resource	4GB
<code>ollama_chat/mistral</code>	7B	Balanced speed/quality	8GB
<code>ollama_chat/phi4</code>	14B	Coding, STEM	16GB
<code>ollama_chat/gemma2</code>	9B	Google, instruction following	12GB
<code>ollama_chat/qwen2.5</code>	7B-72B	Multilingual	8-40GB

**Model string format:** `ollama_chat/[model-name]` ⚠ NOT `ollama/` !

## Configuration Options

```
from google.adk.models import LiteLlm

# Basic usage
model = LiteLlm(model='ollama_chat/llama3.3')

# With custom Ollama server
os.environ['OLLAMA_API_BASE'] = 'http://192.168.1.100:11434'
model = LiteLlm(model='ollama_chat/llama3.3')

# With additional parameters (passed to Ollama)
model = LiteLlm(
    model='ollama_chat/llama3.3',
    temperature=0.7,
    top_p=0.9,
    max_tokens=2048
)
```

## 4. Azure OpenAI Integration

**Azure OpenAI** is for enterprises with **Azure contracts** or **compliance requirements**.

### Azure Setup

1. **Create Azure OpenAI resource** in Azure Portal
2. **Deploy a model** (e.g., gpt-4o)
3. **Get credentials:**
  - API key from Azure Portal
  - Endpoint URL (e.g., `https://your-resource.openai.azure.com/`)
  - Deployment name (e.g., `gpt-4o-deployment`)
4. **Set environment variables:**



## | Example: Azure OpenAI Agent

```

"""
ADK agent using Azure OpenAI.
"""

import asyncio
import os
from google.adk.agents import Agent
from google.adk.runners import InMemoryRunner
from google.adk.models import LiteLlm
from google.genai import types

# Azure OpenAI configuration
os.environ['AZURE_API_KEY'] = 'your-azure-key'
os.environ['AZURE_API_BASE'] = 'https://your-resource.openai.azure.com/'
os.environ['AZURE_API_VERSION'] = '2024-02-15-preview'

async def main():
    """Agent using Azure OpenAI."""

    # Create LiteLLM model - format: "azure/deployment-name"
    azure_model = LiteLlm(model='azure/gpt-4o-deployment')

    # Create agent
    agent = Agent(
        model=azure_model,
        name='azure_agent',
        description='Agent using Azure OpenAI',
        instruction='You are an enterprise assistant running on Azure.'
    )

    # Create runner and session
    runner = InMemoryRunner(agent=agent, app_name='azure_app')
    session = await runner.session_service.create_session(
        app_name='azure_app',
        user_id='user_001'
    )

    # Run query with async iteration
    query = "Explain the benefits of Azure OpenAI for enterprises"
    new_message = types.Content(
        role='user',
        parts=[types.Part(text=query)]
    )

    async for event in runner.run_async(
        user_id='user_001',
        session_id=session.id,

```

```
        new_message=new_message
    ):
        if event.content and event.content.parts:
            print(event.content.parts[0].text)

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

### Why Azure OpenAI?

- ✓ Enterprise SLAs (99.9% uptime)
- ✓ Data residency (EU, US, Asia)
- ✓ Private networks (VNet integration)
- ✓ Compliance (SOC 2, HIPAA, GDPR)
- ✓ Unified billing with Azure services

## 5. Claude via Vertex AI

**Claude on Vertex AI** combines Anthropic's models with Google Cloud infrastructure.

### | Vertex AI Setup

**1. Enable Vertex AI API** in Google Cloud Console

**2. Set up authentication:**



**3. Ensure Vertex AI Claude access** (may require approval)

## | Example: Claude via Vertex AI



```

"""
ADK agent using Claude 3.7 Sonnet via Vertex AI.
"""

import asyncio
import os
from google.adk.agents import Agent
from google.adk.runners import InMemoryRunner
from google.adk.models import LiteLlm
from google.genai import types

# Vertex AI configuration
os.environ['GOOGLE_CLOUD_PROJECT'] = 'your-project'
os.environ['GOOGLE_CLOUD_LOCATION'] = 'us-central1'

async def main():
    """Agent using Claude via Vertex AI."""

    # Create LiteLLM model - format: "vertex_ai/model-name"
    claude_vertex = LiteLlm(model='vertex_ai/claude-3-7-sonnet@20250219')

    # Create agent
    agent = Agent(
        model=claude_vertex,
        name='claude_vertex_agent',
        description='Agent using Claude on Vertex AI',
        instruction='You leverage Claude via Google Cloud infrastructure.'
    )

    # Create runner and session
    runner = InMemoryRunner(agent=agent, app_name='vertex_claude_app')
    session = await runner.session_service.create_session(
        app_name='vertex_claude_app',
        user_id='user_001'
    )

    # Run query with async iteration
    query = "Compare Claude direct vs. Claude on Vertex AI"
    new_message = types.Content(
        role='user',
        parts=[types.Part(text=query)]
    )

    async for event in runner.run_async(
        user_id='user_001',
        session_id=session.id,
        new_message=new_message
    )

```

```
    ):
        if event.content and event.content.parts:
            print(event.content.parts[0].text)

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

Claude Direct vs. Vertex AI:

Factor	Direct (Anthropic)	Via Vertex AI
Pricing	Per-token pricing	Same or slightly higher
Data residency	US-based	Choose GCP region
SLA	Standard	Google Cloud SLA
Integration	Anthropic API	Unified with GCP
Billing	Separate	Unified GCP billing
Setup	Simpler	More complex

Use Vertex AI Claude when:

- ✔ Already using Google Cloud extensively
- ✔ Need data residency in specific GCP regions
- ✔ Want unified GCP billing
- ✔ Require Google Cloud SLAs

## 6. Multi-Provider Comparison

**Use Case:** Compare response quality across multiple providers for the same query.

```

"""
Multi-provider agent comparison.
Test same query across Gemini, GPT-4o, Claude, and Llama 3.3.
"""

import asyncio
import os
from google.adk.agents import Agent, Runner
from google.adk.models import GoogleGenAI, LiteLlm

# Environment setup
os.environ['GOOGLE_GENAI_USE_VERTEXAI'] = '1'
os.environ['GOOGLE_CLOUD_PROJECT'] = 'your-project'
os.environ['GOOGLE_CLOUD_LOCATION'] = 'us-central1'
os.environ['OPENAI_API_KEY'] = 'sk-...'
os.environ['ANTHROPIC_API_KEY'] = 'sk-ant-...'
os.environ['OLLAMA_API_BASE'] = 'http://localhost:11434'

async def compare_models():
    """Compare response quality across 4 providers."""

    # Define models
    models = {
        'Gemini 2.5 Flash': GoogleGenAI(model='gemini-2.5-flash'),
        'GPT-4o': LiteLlm(model='openai/gpt-4o'),
        'Claude 3.7 Sonnet': LiteLlm(model='anthropic/claude-3-7-sonnet-202502'),
        'Llama 3.3 (Local)': LiteLlm(model='ollama_chat/llama3.3')
    }

    # Test query
    query = """
Explain quantum entanglement to a 12-year-old.
Use an analogy they can relate to.
    """.strip()

    print("\n" + "="*70)
    print("MULTI-PROVIDER MODEL COMPARISON")
    print("="*70 + "\n")
    print(f"Query: {query}\n")
    print("="*70 + "\n")

    # Test each model
    for model_name, model in models.items():
        print(f"### {model_name}")
        print("-" * 70)

        agent = Agent(

```

```

        model=model,
        instruction='You explain complex topics clearly and simply.'
    )

    # Create runner and session for this model
    runner = InMemoryRunner(agent=agent, app_name='compare_app')
    session = await runner.session_service.create_session(
        app_name='compare_app',
        user_id='user_001'
    )

    try:
        # Run query with async iteration
        new_message = types.Content(
            role='user',
            parts=[types.Part(text=query)]
        )

        response = ""
        async for event in runner.run_async(
            user_id='user_001',
            session_id=session.id,
            new_message=new_message
        ):
            if event.content and event.content.parts:
                response = event.content.parts[0].text

        print(response)
        print(f"\n[Length: {len(response)} chars]")

    except Exception as e:
        print(f"Error: {e}")

    print("\n" + "="*70 + "\n")

if __name__ == '__main__':
    asyncio.run(compare_models())

```

### Example Output:

---

## MULTI-PROVIDER MODEL COMPARISON

---

Query: Explain quantum entanglement to a 12-year-old.  
Use an analogy they can relate to.

---

### ### Gemini 2.5 Flash

---

Imagine you have two magic coins. When you flip one and it lands on heads, the other coin INSTANTLY lands on tails - no matter how far apart they are. Even if one coin is on Earth and the other is on Mars!

That's quantum entanglement. Two particles become "entangled" so that measuring one INSTANTLY affects the other, even across huge distances.

[Length: 387 chars]

---

### ### GPT-4o

---

Think of quantum entanglement like having two magical dice that are connected. When you roll one die and it shows a 6, the other die automatically shows a 1 - instantly, even if it's on the other side of the world! Scientists don't fully understand HOW this happens, but they know it does. It's one of the strangest things in physics!

[Length: 415 chars]

---

### ### Claude 3.7 Sonnet

---

Imagine you and your best friend each have a magic marble. No matter how far apart you go - even if you go to different countries - when you squeeze your marble and it turns red, your friend's marble turns blue at the EXACT same instant.

That's quantum entanglement! Two particles become linked so that what happens to one immediately affects the other, no matter the distance. Einstein called it "spooky action at a distance" because even he found it weird!

[Length: 512 chars]

=====  
### Llama 3.3 (Local)  
-----

Think of quantum entanglement like having two special coins that are twins. If you flip one coin and it lands on heads, the other coin will always land on tails - instantly! They're connected in a mysterious way that scientists are still trying to fully understand.

[Length: 287 chars]  
=====

### Observations:

- **Gemini 2.5 Flash:** Fast, concise, accurate
- **GPT-4o:** Clear analogy, acknowledges mystery
- **Claude 3.7 Sonnet:** Most detailed, includes Einstein quote
- **Llama 3.3:** Shortest, simpler but less engaging

# 7. Cost Optimization Strategies

## | Cost Comparison (per 1M tokens)

Provider	Model	Input Cost	Output Cost	Total (1M in + 1M out)
Google	gemini-2.5-flash	\$0.075	\$0.30	<b>\$0.375</b> ★ Cheapest
Google	gemini-2.5-pro	\$1.25	\$5.00	\$6.25
OpenAI	gpt-4o-mini	\$0.15	\$0.60	\$0.75
OpenAI	gpt-4o	\$2.50	\$10.00	\$12.50
Anthropic	claude-3-5-haiku	\$0.80	\$4.00	\$4.80
Anthropic	claude-3-7-sonnet	\$3.00	\$15.00	\$18.00
Ollama	granite4:latest (local)	\$0	\$0	<b>\$0</b> 🎉 Free

## Strategy 1: Tiered Model Selection

```
def get_model_for_task(complexity: str):
    """Select model based on task complexity."""

    if complexity == 'simple':
        # Use cheapest model for simple tasks
        return LiteLlm(model='openai/gpt-4o-mini') # Or gemini-2.5-flash

    elif complexity == 'medium':
        # Balanced cost/quality
        return GoogleGenAI(model='gemini-2.5-flash')

    elif complexity == 'complex':
        # Best reasoning, worth the cost
        return LiteLlm(model='anthropic/claude-3-7-sonnet-20250219')

    elif complexity == 'local_ok':
        # Privacy/cost priority
        return LiteLlm(model='ollama_chat/llama3.3')

# Example usage
simple_agent = Agent(model=get_model_for_task('simple'))
complex_agent = Agent(model=get_model_for_task('complex'))
```



## Strategy 2: Fallback Chain

```

async def run_with_fallback(query: str):
    """Try models in order of cost (cheapest first)."""

    models = [
        ('gemini-2.5-flash', GoogleGenAI(model='gemini-2.5-flash')),
        ('gpt-4o-mini', LiteLlm(model='openai/gpt-4o-mini')),
        ('gpt-4o', LiteLlm(model='openai/gpt-4o'))
    ]

    for model_name, model in models:
        try:
            agent = Agent(model=model)
            runner = InMemoryRunner(agent=agent, app_name='fallback_app')
            session = await runner.session_service.create_session(
                app_name='fallback_app',
                user_id='user_001'
            )

            new_message = types.Content(
                role='user',
                parts=[types.Part(text=query)]
            )

            result_text = None
            async for event in runner.run_async(
                user_id='user_001',
                session_id=session.id,
                new_message=new_message
            ):
                if event.content and event.content.parts:
                    result_text = event.content.parts[0].text

            print(f"✓ Success with {model_name}")
            return result_text

        except Exception as e:
            print(f"✗ {model_name} failed: {e}")
            continue

    raise Exception("All models failed")

```

## | Strategy 3: Local for High Volume

```

"""
Use local Ollama for high-volume, simple tasks.
Use cloud models only when needed.
"""

async def process_batch(queries: list[str]):
    """Process many queries cost-effectively."""

    # Local model for bulk processing
    local_model = LiteLlm(model='ollama_chat/llama3.3')
    local_agent = Agent(model=local_model)

    # Cloud model for complex queries
    cloud_model = GoogleGenAI(model='gemini-2.5-flash')
    cloud_agent = Agent(model=cloud_model)

    results = []

    for query in queries:
        # Route by complexity and create appropriate runner
        if is_simple(query):
            # Free local processing
            runner = InMemoryRunner(agent=local_agent, app_name='batch_app')
        else:
            # Use cloud for complex
            runner = InMemoryRunner(agent=cloud_agent, app_name='batch_app')

        # Create session
        session = await runner.session_service.create_session(
            app_name='batch_app',
            user_id='batch_user'
        )

        # Run query with async iteration
        new_message = types.Content(
            role='user',
            parts=[types.Part(text=query)]
        )

        result_text = None
        async for event in runner.run_async(
            user_id='batch_user',
            session_id=session.id,
            new_message=new_message
        ):
            if event.content and event.content.parts:

```

```

        result_text = event.content.parts[0].text

        results.append(result_text)

    return results

def is_simple(query: str) -> bool:
    """Determine if query is simple enough for local model."""
    simple_keywords = ['what is', 'define', 'explain', 'summarize']
    return any(kw in query.lower() for kw in simple_keywords)

```

## 8. Best Practices

### DO

#### 1. Use Native Gemini When Possible:

```

# ✓ BEST - Native Gemini
agent = Agent(model='gemini-2.5-flash')

# ✗ DON'T - Gemini via LiteLLM (slower, missing features)
agent = Agent(model=Litellm(model='gemini/gemini-2.5-flash'))

```

#### 2. Set Environment Variables Securely:

```

import os

# ✓ GOOD - From environment
api_key = os.environ.get('OPENAI_API_KEY')

# ✗ BAD - Hardcoded
api_key = 'sk-...' # Never commit this!

```

#### 3. Handle Provider-Specific Errors:

```

try:
    result = await runner.run_async(query, agent=agent)
except Exception as e:
    if 'rate_limit' in str(e).lower():
        print("Hit rate limit, waiting...")
        await asyncio.sleep(60)
    elif 'quota' in str(e).lower():
        print("Quota exceeded, switching provider...")
        agent.model = fallback_model
    else:
        raise

```

#### 4. Use Ollama Correctly:

```

# ✓ CORRECT - ollama_chat prefix
model = LiteLlm(model='ollama_chat/llama3.3')

# ✗ WRONG - ollama prefix (limited functionality)
model = LiteLlm(model='ollama/llama3.3')

```

#### 5. Monitor Costs:

```

import time

class CostTracker:
    def __init__(self):
        self.total_tokens = 0
        self.model_costs = {
            'openai/gpt-4o': 2.50 / 1_000_000, # per input token
            'anthropic/claude-3-7-sonnet-20250219': 3.00 / 1_000_000
        }


    def track(self, model: str, tokens: int):
        cost = tokens * self.model_costs.get(model, 0)
        self.total_tokens += tokens
        print(f"Cost: ${cost:.4f} | Total: {self.total_tokens:,} tokens")


tracker = CostTracker()

```


# | DON'T


## 1. Don't Use LiteLLM for Gemini:

```
#  BAD - Loses Gemini-specific features
model = LiteLlm(model='gemini/gemini-2.5-flash')

#  GOOD - Use native
model = 'gemini-2.5-flash' # Or GoogleGenAI('gemini-2.5-flash')
```

## 2. Don't Forget `ollama_chat` Prefix:


```
#  WRONG
LiteLlm(model='ollama/llama3.3')


#  RIGHT
LiteLlm(model='ollama_chat/llama3.3')
```

## 3. Don't Ignore Provider Limits:

- OpenAI: 200K tokens/min (tier dependent)
- Anthropic: 200K tokens/min (varies)
- Ollama: Limited by your GPU

## 4. Don't Mix Credentials:

```
#  BAD - Conflicts

#  GOOD - Use different env names if needed
```

# Summary

You've learned how to use OpenAI, Claude, Ollama, and other LLMs in ADK agents via LiteLLM:

### Key Takeaways:

-  **LiteLLM** enables 100+ LLM providers in ADK

- ✓ **OpenAI:** `LiteLlm(model='openai/gpt-4o-mini')` - requires `OPENAI_API_KEY`
- ✓ **Claude:** `LiteLlm(model='anthropic/claude-3-7-sonnet-20250219')` - requires `ANTHROPIC_API_KEY`
- ✓ **Ollama:** `LiteLlm(model='ollama_chat/granite4:latest')` - ⚠ Use `ollama_chat`, NOT `ollama` !
- ✓ **Azure OpenAI:** `LiteLlm(model='azure/deployment-name')` - enterprise option
- ✓ **DON'T** use LiteLLM for Gemini - use native `GoogleGenAI` instead
- ✓ **Local models** (Ollama) great for privacy, cost, offline use
- ✓ **Cost optimization:** gemini-2.5-flash (\$0.375/1M), gpt-4o-mini (\$0.75/1M), local (free)

### Model String Formats:

Provider	Format	Example
OpenAI	<code>openai/[model]</code>	<code>openai/gpt-4o</code>
Anthropic	<code>anthropic/[model]</code>	<code>anthropic/claude-3-7-sonnet-20250219</code>
Ollama	<code>ollama_chat/[model]</code>	<code>ollama_chat/granite4:latest</code> ⚠ NOT <code>ollama/</code>
Azure	<code>azure/[deployment]</code>	<code>azure/gpt-4o-deployment</code>
Vertex AI	<code>vertex_ai/[model]</code>	<code>vertex_ai/claude-3-7-sonnet@20250219</code>

### When to Use What:

Use Case	Recommended Model
Simple tasks, high volume	gemini-2.5-flash or gpt-4o-mini
Complex reasoning	claude-3-7-sonnet or gpt-4o
Privacy/compliance	ollama_chat/granite4:latest (local)
Enterprise Azure	azure/gpt-4o-deployment
Cost optimization	gemini-2.5-flash (cheapest cloud)
Offline/air-gapped	ollama_chat models
Coding tasks	ollama_chat/phi4 or gpt-4o
Long-form content	claude-3-7-sonnet

### Environment Variables Required:

```
# OpenAI
# Anthropic
# Ollama
# Azure OpenAI
# Google (for native Gemini, not LiteLLM)
```

### Production Checklist:

- [ ] Environment variables configured securely (not hardcoded)
- [ ] API keys stored in secret manager (production)
- [ ] Cost tracking implemented
- [ ] Rate limit handling in place
- [ ] Fallback models configured
- [ ] Ollama models use `ollama_chat` prefix (not `ollama`)
- [ ] NOT using LiteLLM for Gemini (use native instead)
- [ ] Error handling for provider-specific issues
- [ ] Model selection based on task complexity



- [ ] Monitoring and alerting set up

**Next Steps:**

- **Tutorial 22:** Review native Gemini 2.5 models and features
- **Tutorial 26:** Deploy agents to Google AgentSpace
- **Tutorial 27:** Integrate LangChain and CrewAI tools
- **Tutorial 18:** Master Events & Observability

**Resources:**

- [LiteLLM Documentation](https://docs.litellm.ai/) (https://docs.litellm.ai/)
  - [OpenAI API Reference](https://platform.openai.com/docs/api-reference) (https://platform.openai.com/docs/api-reference)
  - [Anthropic Claude Documentation](https://docs.anthropic.com/) (https://docs.anthropic.com/)
  - [Ollama Models](https://ollama.com/library) (https://ollama.com/library)
  - [Azure OpenAI](https://azure.microsoft.com/en-us/products/ai-services/openai-service) (https://azure.microsoft.com/en-us/products/ai-services/openai-service)
  - [ADK LiteLLM Sample](https://github.com/google/adk-docs/tree/main/contributing/samples/hello_world_litellm) (https://github.com/google/adk-docs/tree/main/contributing/samples/hello\_world\_litellm)
- 

**Congratulations!** You can now use OpenAI, Claude, Ollama, and other LLMs in your ADK agents, and you understand when to use native Gemini vs. LiteLLM providers.

---

Generated on 2025-10-21 09:02:57 from 28\_using\_other\_llms.md

Source: Google ADK Training Hub