Creating an AI agent using **LangChain** and the **Gemini** Large Language Model (LLM) involves setting up the environment, defining a tool for the agent to use, initializing the Gemini model, and finally creating and running the agent.

The general steps for creating a basic LangChain agent with Gemini are:

□ Prerequisites & Setup

1. **Get a Gemini API Key:** You'll need a Google AI API key. Set this as an environment variable named `GEMINI_API_KEY`.
2. **Install Packages:** Install the necessary Python packages for LangChain and the Gemini integration.

Bash

```
pip install langchain langchain-google-genai langchain-community
```

□ Core Agent Components

A basic LangChain agent comprises three essential components:

1. **The LLM:** The "brain" that reasons and decides what to do. This will be a Gemini model.
2. **Tools:** External functions the agent can call to get up-to-date information, perform calculations, or interact with external systems.
3. **An Agent and Agent Executor:** The logic that orchestrates the LLM's decisions and executes the chosen tools (the "body").

□ Step-by-Step Implementation

1. Initialize the Gemini LLM

Use the `ChatGoogleGenerativeAI` class from the `langchain-google-genai` package to initialize your model. It's best practice to use a powerful model like `gemini-2.5-pro` or `gemini-2.5-flash` for agentic workflows due to their strong function-calling/tool-use capabilities.

Python

```
import os
from langchain_google_genai import ChatGoogleGenerativeAI

# Make sure the GEMINI_API_KEY environment variable is set
if "GEMINI_API_KEY" not in os.environ:
```

```
    print("Error: GEMINI_API_KEY environment variable not set.")
    exit()

# Initialize the Gemini LLM
llm = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash",
    temperature=0.0 # Recommended to use low temperature for reliable agent
decisions
)
```

2. Define Tools

Tools are standard Python functions decorated with `@tool` from `langchain_core.tools`. The function name and docstring are automatically used by the model for tool-use planning.

Python

```
from langchain_core.tools import tool

# Define a simple tool
@tool
def multiply_numbers(a: float, b: float) -> float:
    """Multiplies two numbers and returns the result."""
    return a * b

# Define the list of tools available to the agent
tools = [multiply_numbers]
```

3. Create the Agent

LangChain agents are often built using the **ReAct (Reasoning and Acting)** pattern, where the LLM reasons step-by-step before deciding on an action. A common, simple way to create an agent that leverages Gemini's function calling is using the `create_tool_calling_agent` utility.

Python

```
from langchain.agents import create_tool_calling_agent
from langchain_core.prompts import ChatPromptTemplate
from langchain.agents import AgentExecutor

# 1. Define the Agent Prompt
# This system prompt guides the agent's behavior and reasoning process.
prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "You are a helpful and witty assistant who can perform
calculations using the provided tools."),
        ("placeholder", "{chat_history}"), # Optional: for
memory/conversation context
        ("human", "{input}"),
        ("placeholder", "{agent_scratchpad}"),
    ]
)
```

```
# 2. Create the Agent Runnable
# This combines the LLM, the tools, and the prompt into a single agent logic.
agent = create_tool_calling_agent(llm, tools, prompt)

# 3. Create the Agent Executor
# This handles the full loop: LLM decision -> Tool execution -> LLM final
answer.
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)
```

4. Run the Agent

Now you can invoke the agent with a query that requires the use of the defined tool.

Python

```
# The agent will recognize it needs the 'multiply_numbers' tool for this task
result = agent_executor.invoke({"input": "What is 123 multiplied by 45?"})

print("\n--- Agent Response ---")
print(result["output"])
```

When you run this, you will see the `verbose=True` output showing the agent's internal thought process:

1. **LLM Call:** The LLM receives the prompt and decides to call the `multiply_numbers` tool with arguments (123, 45).
2. **Tool Execution:** The `AgentExecutor` executes the Python function.
3. **Observation:** The result (5535) is returned to the LLM.
4. **Final LLM Call:** The LLM receives the observation and uses it to formulate a final, natural language answer.

The **all-in-one script** to create a LangChain agent powered by the Gemini LLM.

This script includes all the necessary components: environment setup, model initialization, tool definition, agent creation, and execution. This approach uses the recommended **Function/Tool Calling** method for Gemini agents.

□ All-in-One LangChain Agent Code

This complete script allows the agent to reason about a task and use a provided tool to solve a math problem.

Python code:-

```
import os
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.tools import tool
from langchain.agents import create_tool_calling_agent, AgentExecutor
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.messages import HumanMessage

# --- 1. ENVIRONMENT SETUP (PRACTICAL REQUIREMENT) ---
# NOTE: Replace 'YOUR_API_KEY' with your actual Gemini API key, or
# preferably, set it as an environment variable (GEMINI_API_KEY).
# For simplicity in an 'all-in-one' example, we'll check the env var.

if "GEMINI_API_KEY" not in os.environ:
    print("--- WARNING ---")
    print("Please set the GEMINI_API_KEY environment variable to run this
script.")
    # Exit or set a dummy value if running in an environment where setting
the key is impossible
    # For a real run, this check is crucial.
    # os.environ["GEMINI_API_KEY"] = "YOUR_API_KEY_HERE"
else:
    print("GEMINI_API_KEY detected. Initializing agent...")
print("-" * 30)

# --- 2. DEFINE TOOLS ---
# Tools are standard Python functions decorated with @tool.
@tool
def calculate_area_of_circle(radius: float) -> float:
    """
    Calculates the area of a circle given its radius.
    Use this tool when a user asks for the area of a circle.
    """
    return 3.14159 * (radius ** 2)

tools = [calculate_area_of_circle]

# --- 3. INITIALIZE GEMINI LLM ---
# Use a strong model like gemini-2.5-flash for its robust tool-use
capabilities.
llm = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash",
    temperature=0.0 # Low temperature is better for reliable agent decisions
)

# --- 4. CREATE THE AGENT ---
# The agent needs a prompt, the LLM, and the tools.

# Define the Agent Prompt (the instructions for the agent's behavior)
prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "You are a specialized AI assistant. Your primary function
is to use your provided tools to answer user questions, specifically for
geometry calculations. Be helpful and clear in your final response."),
        # The {input} and {agent_scratchpad} placeholders are managed by the
AgentExecutor
```

```
        HumanMessage(content="{input}"),
        ("placeholder", "{agent_scratchpad}"),
    ]
)

# Create the Agent Runnable (the core logic)
agent_runnable = create_tool_calling_agent(
    llm=llm,
    tools=tools,
    prompt=prompt,
)

# Create the Agent Executor (the mechanism that runs the decision loop)
agent_executor = AgentExecutor(
    agent=agent_runnable,
    tools=tools,
    verbose=True, # Set to True to see the internal steps (reasoning and tool
use)
)

# --- 5. EXECUTE THE AGENT ---
user_query = "What is the area of a circle with a radius of 7.5 meters?"

print(f"User Query: {user_query}\n")
print("--- Agent Execution Trace (verbose=True) ---")

result = agent_executor.invoke({"input": user_query})

print("\n" + "=" * 30)
print(f"Final Agent Answer: {result['output']}")
print("=" * 30)
```

Key Components Explained

Component	LangChain Class	Description
LLM	`ChatGoogleGenerativeAI`	The "**brain**" that interprets the user's request, reasons about the tools needed, and formulates the final answer.
Tool	`@tool` decorator	The "**hands**" of the agent. A Python function the LLM can decide to call to perform an action (like a calculation, database query, or web search).
Prompt	`ChatPromptTemplate`	The "**instruction manual**" that defines the agent's persona, system rules, and the structure of the conversation.

Component	LangChain Class	Description
Agent Runnable	`create_tool_calling_agent`	The core **logic** that connects the LLM, the tools, and the prompt, allowing the LLM to use tool-calling.
Agent Executor	`AgentExecutor`	The **orchestrator** that manages the entire lifecycle: receiving the user input, running the LLM, executing the tool (if called), feeding the result back to the LLM, and returning the final output.