Lesson 5 — Agent Orchestration and Message Handling

Overview

This lesson teaches you how to orchestrate a complete agentic system using LangChain, Google Gemini, and LangSmith tracing. You'll learn to combine LLMs with custom tools, design effective system prompts, handle agent responses, and implement production-grade observability.

This is the capstone of Module 1: bringing together everything from Lessons 1-4 to create a fully functional agent that:

- Interprets natural language queries
- Calls the right tools at the right time
- Handles tool outputs intelligently
- Provides clear, actionable responses
- Traces all operations for debugging and optimization

Learning Outcomes

After completing this lesson, you will be able to:

1. Create LangChain agents with create_tool_calling_agent
2. Integrate Google Gemini models via langchain-google-genai
3. Write system prompts that guide agent behavior
4. Build AgentExecutor configurations
5. Parse and format agent responses (AIMessage handling)
6. Set up LangSmith tracing for production observability
7. Debug agent behavior with comprehensive logging

Prerequisites

- Completion of Lessons 1-4
- Understanding of LangChain tool patterns (Lesson 2)
- Knowledge of Pydantic schemas and tool creation (Lessons 2-4)
- Familiarity with async/await patterns (helpful but not required)

The Problem: From Tools to Intelligent System

What We've Built So Far

Lesson 2: Pydantic schemas + StructuredTool pattern
Lesson 3: Table filtering tool
Lesson 4: API call generation tool

What's Missing

- **Orchestration**: Who decides when to call which tool?
- **Context**: How does the agent remember previous steps?
- **Interpretation**: How do we turn tool outputs into user-facing answers?
- **Observability**: How do we debug when things go wrong?

The Agent Pattern

```
User Query
    ↓
LLM (Gemini) → Decides: which tool? what parameters?
    ↓
Tool Execution → filter_tables_by_allowlist(...)
    ↓
Tool Output → {"allowed": [...], "rejected": [...]}
    ↓
LLM (Gemini) → Interprets result and responds to user
    ↓
Final Answer: "I found 25 allowed tables: ..."
```

Part 1: LangChain Agent Architecture

Core Components

```python
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain.agents import create_tool_calling_agent, AgentExecutor
from langchain_core.prompts import ChatPromptTemplate

# 1. Language Model
llm = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash",
    temperature=0.0
)

# 2. Tools (from Lessons 2-4)
tools = [
    filter_tables_tool,
    data_request_tool,
]

# 3. Prompt Template
prompt = ChatPromptTemplate.from_messages([
```

```
    ("system", "You are a helpful assistant..."),
    ("human", "{input}"),
    ("placeholder", "{agent_scratchpad}"),
])

# 4. Agent (reasoning engine)
agent = create_tool_calling_agent(
    llm=llm,
    tools=tools,
    prompt=prompt
)

# 5. Executor (runtime)
agent_executor = AgentExecutor(
    agent=agent,
    tools=tools,
    verbose=True
)

# 6. Invoke
response = agent_executor.invoke({"input": "Filter the tables"})
```

Component Responsibilities

Component	Purpose	Configured By
LLM	Reasoning, decision-making, natural language	Model choice, temperature
Tools	Actions the agent can take	StructuredTool definitions
Prompt	Instructions, context, behavior guidelines	System message, examples
Agent	Connects LLM + tools + prompt	create_tool_calling_agent
Executor	Manages execution loop, safety, retries	AgentExecutor config

Part 2: Google Gemini Integration

Model Selection

```
from langchain_google_genai import ChatGoogleGenerativeAI

# Available Gemini models (as of January 2025)
models = {
    "gemini-2.5-flash": {
        "speed": "fastest",
        "cost": "lowest",
        "context": "1M tokens",
        "use_case": "development, prototyping, high-throughput"
    },
    "gemini-1.5-pro": {
        "speed": "moderate",
```

```python
        "cost": "moderate",
        "context": "2M tokens",
        "use_case": "production, complex reasoning"
    },
    "gemini-1.5-flash": {
        "speed": "fast",
        "cost": "low",
        "context": "1M tokens",
        "use_case": "balanced performance/cost"
    },
}

# For our use case (Campaign 1 analysis):
llm = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash",       # Latest, fastest
    temperature=0.0,                # Deterministic (recommended for numeric/tool tasks)
    max_tokens=None,                # Use model default
    timeout=None,                   # No timeout
)
```

Why Gemini 2.5 Flash?

1. Speed: Fastest model family, ideal for tool-calling workflows
2. Cost: Most economical for development
3. Context: 1M tokens is more than enough for our Campaign 1 data
4. Tool calling: Excellent native support for function calling
5. Latest: Incorporates newest improvements (Jan 2025 release)

Temperature Setting

```python
# Temperature: Controls randomness

# temperature=0.0 (Deterministic)
#  Use for: Tool selection, structured output, calculations
#  Avoid for: Creative writing, brainstorming

llm_deterministic = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash",
    temperature=0.0  # Same input → same output
)

# temperature=0.7-1.0 (Creative)
# Use for: Varied explanations, multiple phrasings
# Avoid for: Precise tool calling

llm_creative = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash",
    temperature=0.7  # Same input → varied outputs
)
```

For databreeders agent: Always use `temperature=0.0`

- **Reason**: Tool selection must be deterministic
- **Benefit**: Reproducible behavior for testing and debugging

Part 3: System Prompts - Guiding Agent Behavior

Anatomy of a Good System Prompt

From `lesson_01_working_code.py` (lines 560-595):

```
system_message = """You are a helpful assistant for analyzing TV advertising campaign data from dataB

Your role:
1. Help users filter and select the right tables from Campaign 1 data
2. Generate mock API calls to fetch campaign performance data
3. Provide clear explanations of what data is available and how to access it

Guidelines:
- When a user asks about tables or filtering, use the filter_tables_by_allowlist tool
- When a user asks for specific campaign data (reach, frequency, demographics), use the generate_mock_
- Always explain what you're doing and why
- If a tool returns an error, explain it clearly and suggest alternatives
- Be concise but informative

Available data types:
- 'impacts_by_sex_age': Reach and frequency by demographics (age, gender)
- 'impacts_in_target': Performance metrics for target audience
- 'r1plus_in_target_buildup': Daily accumulation of reach (1+ exposures)
- 'rf_in_target_overall': Reach and frequency distribution
- 'tv_spot_schedule': Schedule of TV ad airings
- 'universe_by_sex_age': Population demographics
- 'target_universe': Target audience definition
- 'json_request': Campaign metadata

Filtering rules:
- INCLUDE: TRP tables (contain 'trp')
- INCLUDE: TabSummary tables (contain 'tabsummary')
- EXCLUDE: Plot visualizations (contain 'plot')
- EXCLUDE: 30-second equivalent metrics (contain '30eq')
"""
```

System Prompt Best Practices

DO: Be Specific About Roles

```
# Good
"You are a helpful assistant for analyzing TV advertising campaign data from dataBreeders."

# Bad (too vague)
"You are a helpful assistant."
```

DO: Enumerate Tools and When to Use Them

```
# Good
"""
- When a user asks about tables or filtering, use the filter_tables_by_allowlist tool
- When a user asks for specific campaign data, use the generate_mock_api_call tool
"""

# Bad (agent has to guess)
"Use the tools when appropriate."
```

DO: Provide Domain Knowledge

```
# Good
"""
Available data types:
- 'impacts_by_sex_age': Reach and frequency by demographics
- 'impacts_in_target': Performance metrics for target audience
...
"""

# Bad (agent has to learn from trial-and-error)
"Figure out which data types are available."
```

DO: Include Error Handling Guidance

```
# Good
"If a tool returns an error, explain it clearly and suggest alternatives"

# Bad (no guidance on failures)
(No error handling instructions)
```

DO: Set Tone and Style

```
# Good
"Be concise but informative. Always explain what you're doing and why."

# Bad (inconsistent responses)
(No style guidance)
```

DON'T: Make the Prompt Too Long

```
# Bad (overwhelming)
system_message = """..."""  # 5000+ words of instructions

# Good (focused, 200-500 words)
system_message = """..."""  # Clear, scannable sections
```

Part 4: Creating the Agent

Complete Implementation (lines 527-595)

```
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain.agents import create_tool_calling_agent, AgentExecutor
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.tools import StructuredTool
import logging

logger = logging.getLogger(__name__)

def create_simple_agent(
    tools: list[StructuredTool],
    model_name: str = "gemini-2.5-flash",
    temperature: float = 0.0,
    verbose: bool = True,
) -> AgentExecutor:
    """
    Create a LangChain agent with Google Gemini and custom tools.

    Args:
        tools: List of StructuredTool instances to give the agent
        model_name: Gemini model to use (default: gemini-2.5-flash)
        temperature: Sampling temperature (0.0 = deterministic, 1.0 = creative)
        verbose: Enable detailed logging of agent steps

    Returns:
        AgentExecutor ready to invoke with user queries
    """
    logger.info(f"Creating agent with model: {model_name}, tools: {len(tools)}")

    # Step 1: Initialize LLM
    llm = ChatGoogleGenerativeAI(
        model=model_name,
        temperature=temperature,
    )

    # Step 2: Define system prompt
    system_message = """You are a helpful assistant for analyzing TV advertising campaign data from da

Your role:
1. Help users filter and select the right tables from Campaign 1 data
2. Generate mock API calls to fetch campaign performance data
3. Provide clear explanations of what data is available and how to access it

Guidelines:
- When a user asks about tables or filtering, use the filter_tables_by_allowlist tool
- When a user asks for specific campaign data (reach, frequency, demographics), use the generate_mock_
- Always explain what you're doing and why
- If a tool returns an error, explain it clearly and suggest alternatives
- Be concise but informative

Available data types:
- 'impacts_by_sex_age': Reach and frequency by demographics (age, gender)
- 'impacts_in_target': Performance metrics for target audience
- 'r1plus_in_target_buildup': Daily accumulation of reach (1+ exposures)
- 'rf_in_target_overall': Reach and frequency distribution
- 'tv_spot_schedule': Schedule of TV ad airings
- 'universe_by_sex_age': Population demographics
- 'target_universe': Target audience definition
```

```
- 'json_request': Campaign metadata

Filtering rules:
- INCLUDE: TRP tables (contain 'trp')
- INCLUDE: TabSummary tables (contain 'tabsummary')
- EXCLUDE: Plot visualizations (contain 'plot')
- EXCLUDE: 30-second equivalent metrics (contain '30eq')
"""

    # Step 3: Create prompt template
    prompt = ChatPromptTemplate.from_messages([
        ("system", system_message),
        ("human", "{input}"),
        ("placeholder", "{agent_scratchpad}"),
    ])

    # Step 4: Create agent
    agent = create_tool_calling_agent(
        llm=llm,
        tools=tools,
        prompt=prompt,
    )

    # Step 5: Create executor
    agent_executor = AgentExecutor(
        agent=agent,
        tools=tools,
        verbose=verbose,
        max_iterations=10,           # Prevent infinite loops
        early_stopping_method="generate",  # Return partial results on max iterations
        handle_parsing_errors=True,  # Gracefully handle LLM output errors
    )

    logger.info("Agent created successfully")
    return agent_executor
```

Step-by-Step Breakdown

Step 1: Initialize LLM

```
llm = ChatGoogleGenerativeAI(
    model=model_name,
    temperature=temperature,
)
```

Key decisions:

- Model: Parameterized (default: `gemini-2.5-flash`)
- Temperature: Parameterized (default: `0.0` for determinism)

Step 2: Define System Prompt

```
system_message = """You are a helpful assistant..."""
```

Content:

- Role definition
- Task enumeration
- Tool usage guidelines
- Domain knowledge (data types, filtering rules)
- Error handling instructions
- Style guidance

Step 3: Create Prompt Template

```
prompt = ChatPromptTemplate.from_messages([
    ("system", system_message),
    ("human", "{input}"),
    ("placeholder", "{agent_scratchpad}"),
])
```

Message types:

- `system` : Instructions (constant across invocations)
- `human` : User query (variable, passed via `invoke({"input": "..."})`)
- `placeholder` : Agent's internal reasoning/tool calls (managed by LangChain)

agent_scratchpad: Critical for multi-turn reasoning

- Stores: Previous tool calls, tool outputs, intermediate thoughts
- Enables: "Call tool A, use result in tool B" workflows

Step 4: Create Agent

```
agent = create_tool_calling_agent(
    llm=llm,
    tools=tools,
    prompt=prompt,
)
```

What this does:

- Binds tools to LLM (registers tool schemas)
- Configures tool-calling format (Gemini's native format)
- Sets up reasoning loop

Step 5: Create Executor

```
agent_executor = AgentExecutor(
    agent=agent,
    tools=tools,
    verbose=verbose,
    max_iterations=10,
    early_stopping_method="generate",
```

```
    handle_parsing_errors=True,
)
```

Configuration options:

Parameter	Value	Purpose
`max_iterations`	10	Prevent runaway loops (agent calls tools endlessly)
`early_stopping_method`	`"generate"`	Return partial result if max_iterations hit
`handle_parsing_errors`	`True`	Catch malformed LLM outputs gracefully
`verbose`	`True`	Print step-by-step execution logs

Part 5: Invoking the Agent and Handling Responses

Basic Invocation

```
# Create agent
agent_executor = create_simple_agent(tools=[filter_tool, data_tool])

# Invoke with user query
response = agent_executor.invoke({
    "input": "Filter the tables from element_config.json"
})

# Response structure:
{
    "input": "Filter the tables from element_config.json",
    "output": "I filtered the tables and found 25 allowed tables: ..."
}
```

Response Types

Type 1: Direct Answer (no tools needed)

```
response = agent_executor.invoke({
    "input": "What is dataBreeders?"
})

# Agent reasoning:
# - No tool needed (general knowledge question)
# - Responds directly

print(response["output"])
# "dataBreeders is a TV advertising analytics platform..."
```

Type 2: Single Tool Call

```python
response = agent_executor.invoke({
    "input": "Show me reach by demographics"
})

# Agent reasoning:
# 1. User wants campaign data → use generate_mock_api_call
# 2. Demographics → data_type = "impacts_by_sex_age"
# 3. No audience specified → target_audience = "all"
# 4. Call tool
# 5. Interpret result and respond

print(response["output"])
# "I've generated an API call to fetch reach by demographics..."
```

Type 3: Multi-Step Reasoning (multiple tool calls)

```python
response = agent_executor.invoke({
    "input": "Filter the tables and then get reach data for women"
})

# Agent reasoning:
# 1. First: filter tables → call filter_tables_by_allowlist
# 2. Tool output: 25 allowed tables
# 3. Second: get reach for women → call generate_mock_api_call with target="women_25_34"
# 4. Synthesize both results

print(response["output"])
# "I filtered the tables (25 allowed) and generated an API call for women's reach data..."
```

Part 6: AIMessage Handling and Formatting

Understanding AIMessage

When the agent responds, it returns an `AIMessage` object:

```python
from langchain_core.messages import AIMessage

# Simplified structure
ai_message = AIMessage(
    content="I've filtered the tables...",  # Natural language response
    additional_kwargs={                      # Tool calls, metadata
        "tool_calls": [...],
    }
)
```

Extracting the Response (lines 735-742)

```python
def format_agent_response(response: dict) -> str:
    """
```

```python
    Extract and format the agent's response.

    Args:
        response: Agent executor output (dict with "input" and "output" keys)

    Returns:
        Formatted string response
    """
    # Extract output
    output = response.get("output", "")

    # If output is an AIMessage, extract content
    if hasattr(output, "content"):
        return output.content

    # Otherwise, return as string
    return str(output)

# Usage
response = agent_executor.invoke({"input": "Filter tables"})
formatted = format_agent_response(response)
print(formatted)
```

Rich Response Formatting

```python
def format_agent_response_rich(response: dict) -> dict:
    """Extract and structure agent response components."""

    output = response.get("output", "")

    # Extract text content
    if hasattr(output, "content"):
        text_content = output.content
    else:
        text_content = str(output)

    # Extract tool calls (if any)
    tool_calls = []
    if hasattr(output, "additional_kwargs"):
        tool_calls = output.additional_kwargs.get("tool_calls", [])

    return {
        "text": text_content,
        "tool_calls": tool_calls,
        "input": response.get("input"),
    }

# Usage
response = agent_executor.invoke({"input": "Get reach data"})
formatted = format_agent_response_rich(response)

print(f"User: {formatted['input']}")
print(f"Agent: {formatted['text']}")
print(f"Tools used: {len(formatted['tool_calls'])}")
```

Part 7: LangSmith Tracing

Why Tracing?

Production agentic systems are complex:

- **Multiple steps**: Agent may call 5+ tools per query
- **Non-deterministic**: LLM outputs vary
- **Debugging**: Hard to see "why did the agent do that?"
- **Optimization**: Need metrics to improve performance

LangSmith provides:

- Visual trace trees
- Latency metrics per step
- Input/output capture for every LLM call and tool
- Error tracking
- Comparison across runs

Setup (from lesson_01_working_code.py, lines 29-45)

```
import os
from databreeders_agent.tracing import setup_tracing

def setup_tracing(campaign_name: str, allowlist: str) -> None:
    """
    Configure LangSmith tracing if API key is available.

    Args:
        campaign_name: Name of the campaign (e.g., "Campaign 1")
        allowlist: Allowlist description (e.g., "TRP+TabSummary")
    """
    api_key = os.getenv("LANGSMITH_API_KEY")

    if not api_key:
        print("⚠️  LANGSMITH_API_KEY not set - tracing disabled")
        return

    # Enable tracing
    os.environ["LANGCHAIN_TRACING_V2"] = "true"
    os.environ["LANGCHAIN_PROJECT"] = f"databreeders-{campaign_name}"

    # Set metadata
    os.environ["LANGCHAIN_METADATA"] = json.dumps({
        "campaign": campaign_name,
        "allowlist": allowlist,
        "timestamp": datetime.now().isoformat(),
    })

    print(f" LangSmith tracing enabled: project='databreeders-{campaign_name}'")

# Usage
```

```
setup_tracing(campaign_name="Campaign 1", allowlist="TRP+TabSummary")
```

Environment Variables

```
# Required
export LANGSMITH_API_KEY="lsv2_pt_..."

# Automatically set by setup_tracing():
# LANGCHAIN_TRACING_V2=true
# LANGCHAIN_PROJECT=databreeders-Campaign 1
# LANGCHAIN_METADATA={"campaign": "Campaign 1", ...}
```

Viewing Traces

1. Go to https://smith.langchain.com
2. Navigate to your project: `databreeders-Campaign 1`
3. Click on a trace to see:
    - **Timeline**: When each step occurred
    - **LLM calls**: Inputs, outputs, token counts
    - **Tool calls**: Arguments, return values
    - **Errors**: Stack traces, error messages
    - **Metadata**: Campaign, allowlist, timestamp

Trace Structure Example

```
Query: "Filter tables and get reach data"
└── Agent Executor
    ├── LLM Call 1 (Gemini 2.0 Flash)
    │   ├── Input: system prompt + user query
    │   └── Output: tool_call = filter_tables_by_allowlist(...)
    ├── Tool: filter_tables_by_allowlist
    │   ├── Input: {...}
    │   └── Output: {"allowed": 25, "rejected": 10, ...}
    ├── LLM Call 2 (Gemini 2.0 Flash)
    │   ├── Input: previous context + tool result
    │   └── Output: tool_call = generate_mock_api_call(...)
    ├── Tool: generate_mock_api_call
    │   ├── Input: {data_type: "impacts_by_sex_age", ...}
    │   └── Output: {success: true, curl_command: "...", ...}
    └── LLM Call 3 (Gemini 2.0 Flash)
        ├── Input: all context + both tool results
        └── Output: "I filtered the tables (25 allowed) and generated..."
```

Debugging with Traces

Problem: Agent calls wrong tool

1. Open trace in LangSmith
2. Find LLM Call 1 (first decision)

3. Check:
    - **Input**: Did system prompt include correct instructions?
    - **Output**: Why did LLM choose this tool?
    - **Tool schema**: Is the tool description clear?

Problem: Tool returns error

1. Find Tool Call in trace
2. Check:
    - **Input args**: Are they valid?
    - **Output**: What's the error message?
    - **Tool code**: Is there a bug?

Problem: Slow response

1. View timeline
2. Identify bottleneck:
    - LLM calls (reduce prompt length, use faster model)
    - Tool execution (optimize tool code)
    - Network latency (cache, parallelize)

Part 8: Complete End-to-End Example

Full Working Code

```
import os
from pathlib import Path
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain.agents import create_tool_calling_agent, AgentExecutor
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.tools import StructuredTool
from databreeders_agent.tracing import setup_tracing
from databreeders_agent.tools import (
    filter_tables_tool,
    data_request_tool,
)

def main():
    """Main entry point for the databreeders agent."""

    # Step 1: Setup tracing
    setup_tracing(campaign_name="Campaign 1", allowlist="TRP+TabSummary")

    # Step 2: Load tools
    tools = [
        filter_tables_tool,
        data_request_tool,
    ]

    # Step 3: Create agent
    agent_executor = create_simple_agent(
```

```python
        tools=tools,
        model_name="gemini-2.5-flash",
        temperature=0.0,
        verbose=True,
    )

    # Step 4: Run agent on sample queries
    queries = [
        "Filter the tables from element_config.json",
        "Get reach by demographics for women 25-34",
        "Show me the TV spot schedule",
    ]

    for query in queries:
        print(f"\n{'='*60}")
        print(f"User: {query}")
        print(f"{'='*60}")

        # Invoke agent
        response = agent_executor.invoke({"input": query})

        # Format and display
        formatted = format_agent_response(response)
        print(f"\nAgent: {formatted}\n")

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

Sample Output

```
============================================================
User: Filter the tables from element_config.json
============================================================

> Entering new AgentExecutor chain...

Invoking: `filter_tables_by_allowlist` with `{'table_names': ['standard_tabcontacts_table_contact_sexa

{'allowed_tables': [...], 'rejected_tables': [...], 'statistics': {...}}

I've filtered the tables from element_config.json. Here's what I found:

**Allowed tables:** 25 tables passed the allowlist rules
**Rejected tables:** 10 tables were excluded

The allowed tables include TRP tables and TabSummary tables. Rejected tables contained plot visualiza

> Finished chain.

============================================================
User: Get reach by demographics for women 25-34
============================================================

> Entering new AgentExecutor chain...

Invoking: `generate_mock_api_call` with `{'data_type': 'impacts_by_sex_age', 'target_audience': 'womer

{'success': True, 'request_body': {...}, 'curl_command': 'curl -X POST ...'}
```

````
I've generated an API call to fetch reach by demographics for women aged 25-34:

**Request body:**
```json
{
  "campaign_id": "Campaign 1",
  "data_type": "impacts_by_sex_age",
  "target_audience": "women_25_34"
}
````

Curl command:

```
curl -X POST https://api.databreeders.io/v1/campaigns/analyze \
  -H 'Content-Type: application/json' \
  -H 'Authorization: Bearer YOUR_API_KEY' \
  -d '{"campaign_id": "Campaign 1", ...}'
```

> Finished chain.

````
---

## Part 9: Advanced Patterns

### Pattern 1: Streaming Responses

```python
from langchain_core.callbacks import StreamingStdOutCallbackHandler

# Create agent with streaming
llm = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash",
    temperature=0.0,
    streaming=True,
    callbacks=[StreamingStdOutCallbackHandler()],
)

agent_executor = create_simple_agent(tools=tools)

# Response streams to console in real-time
response = agent_executor.invoke({"input": "Filter tables"})
````

Pattern 2: Custom Callbacks for Logging

```
from langchain_core.callbacks import BaseCallbackHandler

class CustomLogCallback(BaseCallbackHandler):
    """Log all LLM and tool events."""

    def on_llm_start(self, serialized, prompts, **kwargs):
```

```python
        print(f"LLM started with {len(prompts)} prompts")

    def on_llm_end(self, response, **kwargs):
        print(f"LLM finished")

    def on_tool_start(self, serialized, input_str, **kwargs):
        print(f"Tool started: {serialized.get('name')}")

    def on_tool_end(self, output, **kwargs):
        print(f"Tool finished")

    def on_tool_error(self, error, **kwargs):
        print(f"Tool error: {error}")

# Use callback
agent_executor = AgentExecutor(
    agent=agent,
    tools=tools,
    callbacks=[CustomLogCallback()],
)
```

Pattern 3: Agent with Memory (Multi-Turn Conversations)

```python
from langchain.memory import ConversationBufferMemory

# Add memory
memory = ConversationBufferMemory(
    memory_key="chat_history",
    return_messages=True,
)

# Update prompt to include history
prompt = ChatPromptTemplate.from_messages([
    ("system", system_message),
    ("placeholder", "{chat_history}"),
    ("human", "{input}"),
    ("placeholder", "{agent_scratchpad}"),
])

agent_executor = AgentExecutor(
    agent=agent,
    tools=tools,
    memory=memory,
)

# Multi-turn conversation
response1 = agent_executor.invoke({"input": "Filter tables"})
response2 = agent_executor.invoke({"input": "Now get reach data"})
# Agent remembers the filtering step from response1
```

Part 10: Hands-On Exercise

Exercise: Add Tool Usage Statistics

Objective: Track which tools the agent uses most frequently.

Requirements:

1. Create a callback that counts tool invocations
2. Track:
   - Total tool calls
   - Calls per tool name
   - Success/failure rates
3. Print summary after agent execution

Starter Code:

```python
from langchain_core.callbacks import BaseCallbackHandler
from collections import defaultdict

class ToolStatsCallback(BaseCallbackHandler):
    """Track tool usage statistics."""

    def __init__(self):
        self.stats = {
            "total_calls": 0,
            "by_tool": defaultdict(int),
            "errors": 0,
        }

    def on_tool_start(self, serialized, input_str, **kwargs):
        # TODO: Increment counters
        pass

    def on_tool_error(self, error, **kwargs):
        # TODO: Track errors
        pass

    def print_summary(self):
        # TODO: Print formatted statistics
        pass

# Usage
stats_callback = ToolStatsCallback()
agent_executor = AgentExecutor(
    agent=agent,
    tools=tools,
    callbacks=[stats_callback],
)

response = agent_executor.invoke({"input": "Filter and get reach data"})
stats_callback.print_summary()
```

Solution:

▶ Click to reveal solution

Knowledge Check

Question 1: Agent Components

What are the 5 core components needed to create a LangChain agent?

A) LLM, tools, prompt, agent, database
B) LLM, tools, prompt, agent, executor
C) LLM, memory, prompt, agent, executor
D) LLM, tools, vector store, agent, executor

▶ Answer

Question 2: Temperature Setting

For tool-calling workflows with databreeders, what temperature should you use?

A) 0.0 (deterministic)
B) 0.5 (balanced)
C) 0.7 (creative)
D) 1.0 (maximum randomness)

▶ Answer

Question 3: agent_scratchpad

What is the purpose of `{agent_scratchpad}` in the prompt template?

A) Store user's previous queries
B) Store agent's intermediate reasoning and tool calls
C) Store error messages
D) Store system logs

▶ Answer

Question 4: LangSmith Tracing

What environment variable enables LangSmith tracing?

A) `LANGCHAIN_TRACING=true`
B) `LANGSMITH_ENABLED=true`
C) `LANGCHAIN_TRACING_V2=true`
D) `LANGCHAIN_DEBUG=true`

▶ Answer

Question 5: AgentExecutor Configuration

What does `max_iterations=10` prevent?

A) Tools from being called more than 10 times total
B) LLM from generating more than 10 tokens
C) Agent from running more than 10 seconds
D) Agent from entering infinite reasoning loops

▶ Answer

Summary

In this lesson, you learned:

1. **Agent architecture**: LLM + tools + prompt + agent + executor
2. **Gemini integration**: Model selection, temperature settings, tool calling
3. **System prompts**: Role definition, tool guidelines, domain knowledge, style
4. **Agent creation**: `create_tool_calling_agent` and `AgentExecutor` configuration
5. **Response handling**: Extracting content from AIMessage objects
6. **LangSmith tracing**: Setup, viewing traces, debugging with traces
7. **Production patterns**: Callbacks, streaming, memory, statistics

Congratulations!

You've completed **Module 1: Foundations of Agentic Development**!

You now know how to:

- Set up Python environments for agentic projects (Lesson 1)
- Design Pydantic schemas and LangChain tools (Lesson 2)
- Implement business logic with allowlist filtering (Lesson 3)
- Generate structured API calls from natural language (Lesson 4)
- Orchestrate complete agentic systems with observability (Lesson 5)

Next Module Preview

Module 2: Advanced Agentic Patterns will cover:

- Complex multi-agent systems
- State management with LangGraph
- Error recovery and self-correction
- Evaluation and testing strategies
- Production deployment patterns

References

- LangChain Agents
- create_tool_calling_agent
- Google Gemini Models
- LangSmith Documentation
- LangChain Callbacks
- AgentExecutor Configuration

Code Reference

Complete working implementation: lesson_01_working_code.py (lines 527-786)