Cheat Sheet: Foundations of Tool Calling and Chaining

Estimated time: 20 minutes

1. Tool Calling Fundamentals

Tool calling lets powerful chat models respond to your prompts by requesting the use of specific tools.
Despite the name suggesting direct action by the model, the model only generates the arguments to a tool necessary for tool execution, while running the tool (or not) is up to the user.

2. Tool Calling Workflow

This is how the process unfolds when a chat program uses a tool:

Process Step	Explanation
Tool Setup + Your Question	Provide the definitions for the tools the model can use, along with your question.
Tool/Function	A Python function you've set up for the model to use, allowing it to handle specific tasks beyond its usual knowledge. **Example:** `get_weather("location")` **Example Query:** "What's the weather in Paris?"
Tool Call	The model decides which tool is needed and then asks for it, providing the necessary details. This request is a structured message specifying the tool to use and its input details. **Example:** `get_weather("paris")`
Tool Execution	The application then runs the chosen tool using the details provided by the model. **Example:** Running `get_weather("paris")` might give you `{"temperature": 14}` **Return Value:** A structured output, often a dictionary or JSON-like object.
Result Passing	The tool's answer is sent back to the model. This answer becomes part of the model's working memory, helping it understand the conversation and task context.
Final Response	The program uses the tool's answer to create a helpful, natural-sounding reply for you. **Example:** "It's currently 14°C in Paris."

3. LangChain Tool Creation Methods

All tools built in LangChain follow a basic blueprint called `BaseTool`. It establishes the core functionality and structure that tools need to implement. Both `Tool` and `StructuredTool` classes inherit from `BaseTool`, making it the foundation of the tool hierarchy.

1. Tool Creation by Subclassing BaseTool Class

Subclassing from `BaseTool` is the most flexible method, providing the largest degree of control at the expense of more effort and code. This approach allows you to define custom instance variables, implement both synchronous and asynchronous methods, and have complete control over tool behavior.

```
from langchain.tools import BaseTool
from typing import Optional
from langchain.callbacks.manager import CallbackManagerForToolRun,
 AsyncCallbackManagerForToolRun

class CustomCalculatorTool(BaseTool):
    name = "Calculator"
    description = "Useful for mathematical calculations"

    def _run(self, query: str, run_manager:
 Optional[CallbackManagerForToolRun] = None) -> str:
        """Use the tool synchronously"""
        # Custom logic here
        return "Calculation result"
    async def _arun(self, query: str, run_manager:
 Optional[AsyncCallbackManagerForToolRun] = None) -> str:
        """Use the tool asynchronously"""
        # Custom async logic here
        return "Async calculation result"
```

2. Tool Creation using the Tool Class

LangChain provides a `Tool` class (which inherits from `BaseTool`) to encapsulate a function along with its metadata, such as name, description, and argument schema. This allows for more control over the tool's behavior and integration. Note: This is the traditional approach that primarily handles single string inputs, though it shares the same base class as `StructuredTool` for backward compatibility.

```python
from langchain.agents import Tool

def add_numbers(a: int, b: int) -> int:
    """Add two numbers."""
    return a + b

add_tool = Tool(
    name="AddTool",
    func=add_numbers,
    description="Adds a list of numbers and returns the result."
```

3. Tool Creation Using the @tool Decorator

The `@tool` decorator is a convenient way to create a tool from a Python function. It automatically infers the tool's name, description, and argument schema from the function's signature and docstring. Tools created using `@tool` decorator create a `StructuredTool` (which inherits from `BaseTool`), which allows LLMs to handle more complex inputs, including named arguments and dictionaries, improving flexibility and integration with function-calling models.

```python
from langchain_core.tools import tool

@tool

def divide_numbers(a: int, b: int) ->
 float:

    """Divide a by b."""
    return a / b
```

4. Structured Tool Creation

`StructuredTool` (which inherits from `BaseTool`) provides the most flexibility for function-based tools and is the modern approach for creating tools that can operate on any number of inputs with arbitrary types. It supports complex argument schemas and both synchronous and asynchronous implementations.

```python
from langchain_core.tools import StructuredTool

def multiply_numbers(a: int, b: int) -> int:
    """Multiply two numbers."""
    return a * b

async def amultiply_numbers(a: int, b: int) -> int:
    """Multiply two numbers asynchronously."""
    return a * b
calculator = StructuredTool.from_function(
    func=multiply_numbers,
    coroutine=amultiply_numbers,
    name="Calculator",
    description="multiply numbers",
    return_direct=True
)
```

Note: The @tool decorator and StructuredTool are the recommended modern approaches for function-based tools, while BaseTool subclassing provides the ultimate flexibility for complex custom behaviors. The Tool class remains for compatibility with existing codebases.

4. Checking & Using Your Tools

Once you've defined your tools, here's how you can inspect and use them:

1. Tool Schema

You can look at a tool's name, description, and what inputs it expects using the following:

```python
print("Name: \n", divide_numbers.name)
print("Description: \n", divide_numbers.description)
print("Args: \n", divide_numbers.args)
```

Output:

```
Name:
 divide_numbers
Description:
 Divide a by b.
Args:
 {'a': {'title': 'A', 'type': 'integer'},
 'b': {'title': 'B', 'type': 'integer'}}
```

2. Direct Invocation of Tools

Tools can be invoked directly using the `invoke()` method by passing a dictionary of arguments. This is especially useful for testing outside of an agent or LLM context.

```
result = divide.invoke({"a": 10, "b": 2})  # Output: 5.0
```

3. Tool Binding to Models

Before a model can use tools, you need to tell it which ones are available and what kind of information they need.

```
tools = [add_tool, divide_numbers]
llm_with_tools = llm.bind_tools(tools)
```

4. Tool Invocation via Model

Once tools are bound to a model, the model can decide to invoke them based on the input prompt. The model's response will include the tool call details, which the application can then execute.

```
response = llm_with_tools.invoke("What is 10 divided by 2?")
```

5. LangChain Built-in Tools

LangChain offers a collection of tools that are ready to use. Always check their official guides, as some might have costs or specific requirements.

Use case	Tool	Purpose
Search	SerpAPI, Wikipedia, Tavily	Web and knowledge search
Math & Code	LLMMathChain, Python REPL, Pandas	Computation, data analysis
Web/API	Requests Toolkit, PlayWright	Web requests, scraping
Productivity	Gmail, Google Calendar, Slack, GitHub	Messaging, scheduling, repo management
Files/Docs	FileSystem, Google Drive, VectorStoreQA	Document access, file ops
Finance	Stripe, Yahoo Finance, Polygon IO	Payments, market data
ML Tools	DALL-E, HuggingFace Hub	Image/gen model interaction

6. Agents in LangChain

Agents are intelligent systems powered by LLMs that use tools, memory, and reasoning logic to perform complex tasks.

Components of an Agent:

- LLM: Core reasoning unit.
- Tools: External functions the LLM can call.
- Memory: Stores conversational or task context.
- Executor: Loops over reasoning steps until a final answer is ready.

Agent Types:

- `zero-shot-react-description`
- `chat-zero-shot-react-description`
- `create_openai_functions_agent`
- LangGraph-based agents

7. LCEL (LangChain Expression Language)

1. **LCEL:** This is a straightforward way to build powerful sequences (called “chains”) by connecting existing parts in LangChain. It uses a simple “pipe” (|) symbol to link them together, which helps LangChain make your programs run faster.
2. **Built Around Runnables:** LCEL is built on a standard way for all LangChain pieces to communicate, allowing you to combine them easily.

3. **Chaining:** This is the process of linking up different parts so that what comes out of one step automatically becomes the input for the next. In LCEL, this is done using the | (pipe) operator, making your AI's workflows clear and easy to read.

4. **Example of a simple LCEL chain:**

```
from langchain_core.runnables import RunnableLambda

# Define individual components
prompt = RunnableLambda(lambda x: f"What is {x}?")
llm = ...  # Some language model
output_parser = RunnableLambda(lambda x: x.upper())

# Chain the components
chain = prompt | llm | output_parser
# Execute the chain
result = chain.invoke("Python")  # Output: LLM generated
 response:"PYTHON"
```