Generative AI

Trainer: Nilesh Ghule

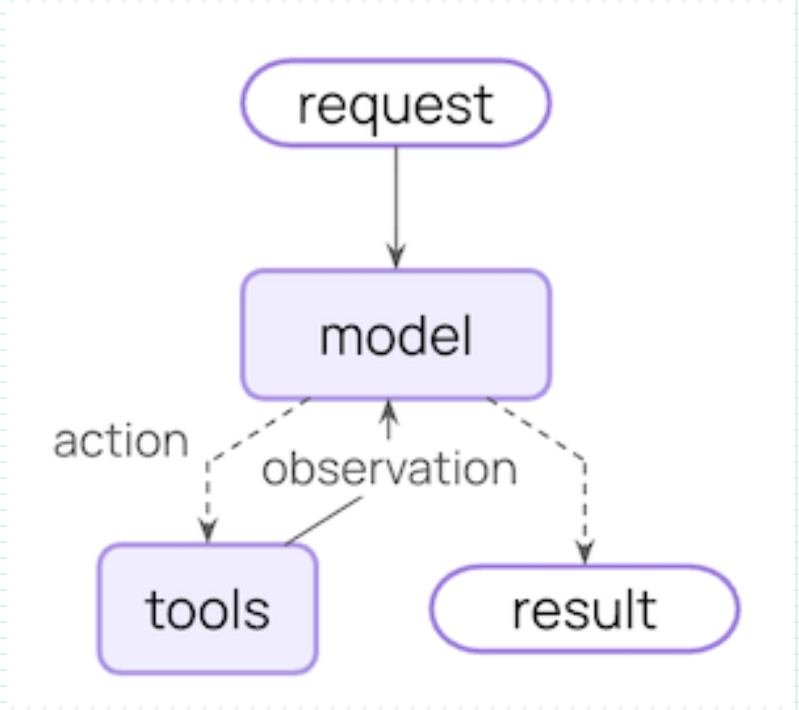
Why LangChain v1? The Redesign

- Before v1: Many abstractions (chains, agents, wrappers, helpers) led to complexity and fragmentation.
- V1 shift: Single unified agent abstraction built on LangGraph internals, deprecating old chains and legacy patterns.
- Goals: Cleaner API, production-ready stability, support for multimodal inputs and structured outputs.

Key Changes in LangChain v1

- Unified agent interface: create_agent() replaces old chain and agent classes.
- Standard content blocks: Text, reasoning, tool calls, and citations handled consistently across providers.
- LangGraph runtime: Enables durable, stateful orchestration under the hood.
- Legacy code moved to langchain-classic for backward compatibility.
- What's new:
  - New standard agent API (create_agent)
  - Middleware system for agent customization
  - Provider-agnostic content blocks
  - Leaner core package
  - Built-in persistence & checkpointing
  - Streaming (tokens, tools, traces)
  - Human-in-the-loop support
  - Production-ready stable v1 baseline

Agent-First Mental Model



- An agent is a loop:
  - invoke model
  - check for tool calls
  - execute tools (if any)
  - repeat until stopping condition.
- Without tools, an agent behaves like a simple chatbot
- With tools, it can search, calculate, scrape, or call APIs dynamically.

Creating a Minimal Agent

- Simplest form:
  - agent = create_agent(model, tools=[])
- Invoke with:
  - agent.invoke({'messages': [{'role': 'user', 'content': 'your prompt'}]})
- Returns a dictionary with 'messages' key containing the full conversation including the model's response.

```
from dotenv import load_dotenv
from langchain.chat_models import init_chat_model
from langchain.agents import create_agent

load_dotenv()

llm = init_chat_model(
        "google/gemma-3-4b",
        model_provider="openai",
        base_url=os.environ.get("OPENAI_BASE_URL"),
        api_key=os.environ.get("OPENAI_API_KEY")
    )

agent = create_agent(model=llm, tools=[])

result = agent.invoke({
    "messages": [{"role": "user", "content": "What is
LangChain"}]
})

print(result["messages"][-1].content)
```

Nilesh Ghule

Agent State and Message History

- Each agent.invoke returns a state dictionary with updated 'messages' list.
- To maintain multi-turn context with agent, pass previous messages into the next invocation.
- For advanced use, LangGraph checkpointing will persist state across sessions automatically.

What Are Tools? Why Tools?

- Tools are callable functions with defined inputs that agents can invoke to perform actions.
- Each tool has: name, description (for LLM decision-making), and input schema (parameters with types).
- Tools extend agent capabilities beyond text generation: calculations, file access, API calls, web scraping.
- Without tools: Agent is limited to training knowledge, cannot access real-time data or perform actions. $\rightarrow$ Simply a chat bot.
- With tools: Agent can search databases, call APIs, read files, scrape websites, and execute code.
- Foundation for: RAG (retrieval tools), Selenium scraping (web tools), multi-agent systems (handoff tools).

Creating Tools with @tool Decorator

- Import: from langchain.tools import tool
- Decorate a function: @tool above the function definition.
- Function name becomes tool name; docstring becomes description; type hints define schema.
- Return type: string, dict, or any JSON-serializable object.

```
from langchain.tools import tool

@tool
def calculator(expression: str) -> str:
    """
    Evaluates a mathematical expression and returns result.
    Supports basic arithmetic: +, -, *, /, parentheses.
    Example: calculator("(10 + 5) * 2") returns "30"
    """
    result = eval(expression)
    return str(result)
```

args & return values of tool fns must be json serializable.

Nilesh Ghule

Code: File Reader Tool

- Example of a tool that reads text file contents.

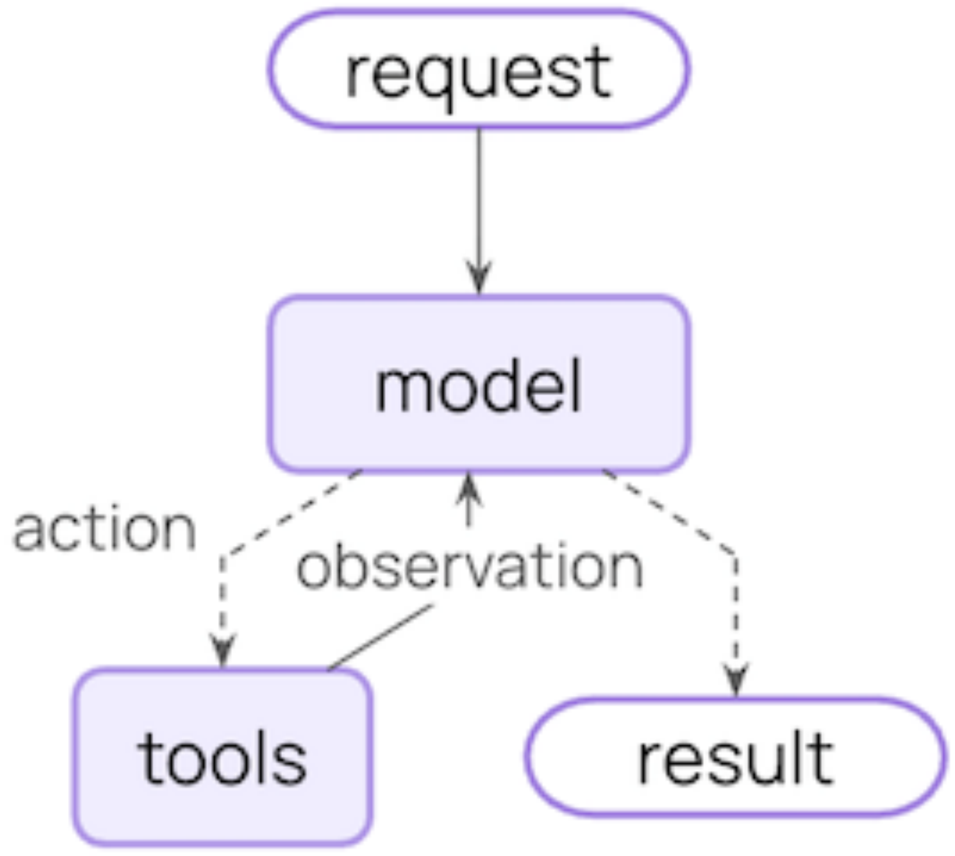
```
from langchain.tools import tool
import os

@tool
def read_file(filename: str) -> str:
    """
    Reads and returns the contents of a text file.
    Filename should be relative to the current working directory.
    Example: read_file("data.txt")
    """
    try:
        if not os.path.exists(filename):
            return f"Error: File '{filename}' not found."
        with open(filename, 'r') as f:
            content = f.read()
        return content
    except Exception as e:
        return f"Error reading file: {str(e)}"
```

How Agents Use Tools?

- To add tools in agent, pass them as a list to create_agent():

  tools=[calculator, read_file, …]
- Model receives user message + tool schemas (names, descriptions, parameters).
- Model can: (1) Answer directly with text, or (2) Request a tool call with specific arguments.
- Agent loop detects tool calls, executes tools, feeds results back to model for final answer.
- Inspect result['messages'] from agent to debug the complete flow.



What is Middleware?

- Middleware: Hooks that intercept and modify the agent loop at key points.
- Use cases: Logging, validation, safety filters, history management, routing, testing.
- Middleware is composable: stack multiple middleware components sequentially.
- Important Middleware Hooks:
  - ✓ before_model: Called before model invocation; can modify request or inject context.
  - ✓ after_model: Called after model response; can validate or filter outputs.
  - ✓ wrap_tool_call: Wraps tool execution; can log, add error handling, or mock responses.
  - ✓ wrap_model_call: Wraps model invocation; can log, etc.

Code: Simple Logging Middleware

- Example middleware.

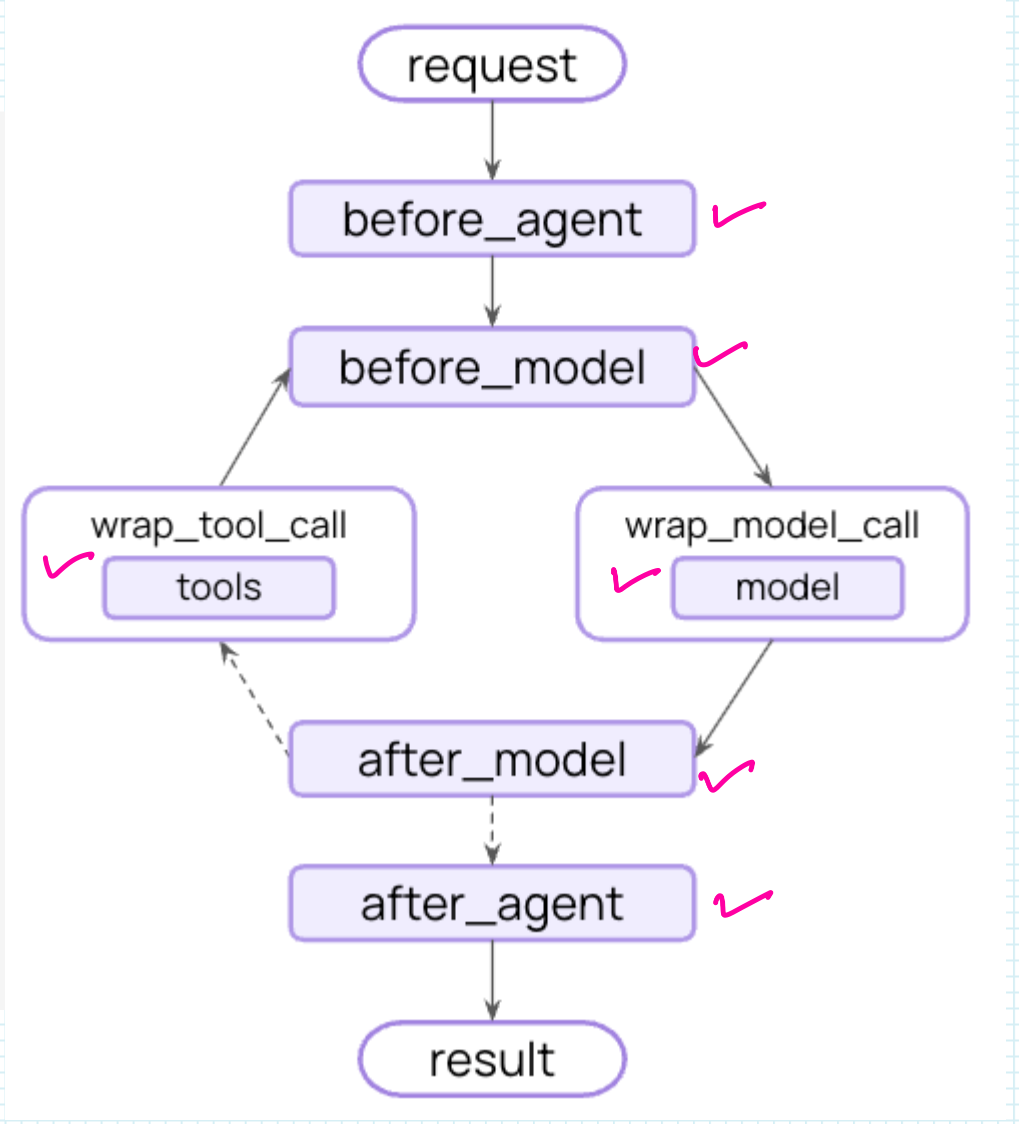
@wrap_model_call

```
def logging_middleware(request, handler):
    """
    Logs each model req and resp for debugging.
    """
    print(f"Req msg count: {len(request.messages)}")

    # Call the next handler in the chain
    response = handler(request)

    print(f"Model responded")
    return response

agent = create_agent(
    model=model,
    tools=[calculator],
    middleware=[logging_middleware]
)
```



Nilesh Ghule

Lab Tasks

- Create tools: calculator, file reader, current weather, and knowledge lookup using @tool decorator.
- Build an agent with all three tools and test with prompts requiring tool usage.
- Inspect message history to understand tool-calling flow.
- Implement a logging middleware and observe its output during agent execution.

Thank You!

Nilesh Ghule

<nilesh@sunbeaminfo.com>