LangChain overview

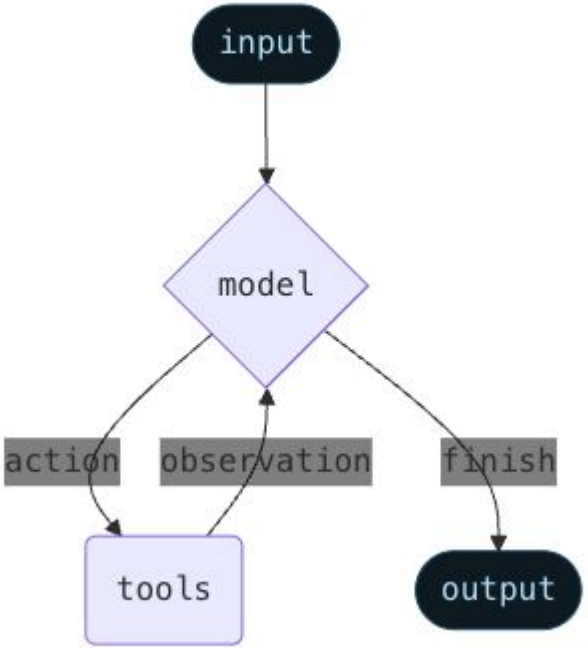
1. Open source framework
2. Pre-built agent architecture
3. Integrations for any model or tool
4. With under 10 lines of code, you can connect to OpenAI, Anthropic, Google, and more
5. Use LangChain if you want to quickly build agents and autonomous applications
6. LangGraph - low-level agent orchestration framework and runtime
7. LangChain agents are built on top of LangGraph in order to provide durable execution, streaming, human-in-the-loop, persistence, and more. You do not need to know LangGraph for basic LangChain agent usage.

Create an agent

```python
# pip install -qU langchain "langchain[anthropic]"

from langchain.agents import create_agent

def get_weather(city: str) -> str:
    """Get weather for a given city."""
    return f"It's always sunny in {city}!"

agent = create_agent(
    model="claude-sonnet-4-5-20250929",
    tools=[get_weather],
    system_prompt="You are a helpful assistant",
)

# Run the agent
agent.invoke(
    {"messages": [{"role": "user", "content": "what is the weather in sf"}]}
)
```

Agents

1. Agents combine language models with tools to create systems that can reason about tasks, decide which tools to use, and iteratively work towards solutions.
2. create_agent provides a production-ready agent implementation.
3. An LLM Agent runs tools in a loop to achieve a goal. An agent runs until a stop condition is met - i.e., when the model emits a final output or an iteration limit is reached.



Models

LLMs are powerful AI tools that can interpret and generate text like humans. They're versatile enough to write content, translate languages, summarize, and answer questions without needing specialized training for each task.

In addition to text generation, many models support:

Tool calling - calling external tools (like databases queries or API calls) and use results in their responses.

Structured output - where the model's response is constrained to follow a defined format.

Multimodality - process and return data other than text, such as images, audio, and video.

Reasoning - models perform multi-step reasoning to arrive at a conclusion.

Messages

Messages are the fundamental unit of context for models in LangChain. They represent the input and output of models, carrying both the content and metadata needed to represent the state of a conversation when interacting with an LLM.

Messages are objects that contain:

Role - Identifies the message type (e.g. system, user)

Content - Represents the actual content of the message (like text, images, audio, documents, etc.)

Metadata - Optional fields such as response information, message IDs, and token usage

LangChain provides a standard message type that works across all model providers, ensuring consistent behavior regardless of the model being called.

Tools

Tools extend what agents can do—letting them fetch real-time data, execute code, query external databases, and take actions in the world.

Under the hood, tools are callable functions with well-defined inputs and outputs that get passed to a chat model. The model decides when to invoke a tool based on the conversation context, and what input arguments to provide.

Create tools

The simplest way to create a tool is with the @tool decorator. By default, the function's docstring becomes the tool's description that helps the model understand when to use it:

Type hints are required as they define the tool's input schema. The docstring should be informative and concise to help the model understand the tool's purpose.

```python
from langchain.tools import tool


@tool
def search_database(query: str, limit: int = 10) -> str:
    """Search the customer database for records matching the query.


    Args:
        query: Search terms to look for
        limit: Maximum number of results to return
    """
    return f"Found {limit} results for '{query}'"
```

Short term memory

Memory is a system that remembers information about previous interactions.

For AI agents, memory is crucial because it lets them remember previous interactions, learn from feedback, and adapt to user preferences.

As agents tackle more complex tasks with numerous user interactions, this capability becomes essential for both efficiency and user satisfaction.

Short term memory lets your application remember previous interactions within a single thread or conversation.

```
from langchain.agents import create_agent
from langgraph.checkpoint.memory
import InMemorySaver


agent = create_agent(
    "gpt-5",
    tools=[get_user_info],
    checkpointer=InMemorySaver(),
)

agent.invoke(
    {"messages": [{"role": "user", "content": "Hi! My
name is Bob."}]},
    {"configurable": {"thread_id": "1"}},
)
```

Streaming

Stream real-time updates from agent runs

LangChain implements a streaming system to surface real-time updates.

Streaming is crucial for enhancing the responsiveness of applications built on LLMs. By displaying output progressively, even before a complete response is ready, streaming significantly improves user experience (UX), particularly when dealing with the latency of LLMs.

What's possible with LangChain streaming:

Stream agent progress — get state updates after each agent step.

Stream LLM tokens — stream language model tokens as they're generated.

Stream custom updates — emit user-defined signals (e.g., "Fetched 10/100 records").

Stream multiple modes — choose from updates (agent progress), messages (LLM tokens + metadata), or custom (arbitrary user data).

Supported streaming modes

updates	Streams state updates after each agent step. If multiple updates are made in the same step (e.g., multiple nodes are run), those updates are streamed separately.
messages	Streams tuples of (token, metadata) from any graph nodes where an LLM is invoked.
custom	Streams custom data from inside your graph nodes using the stream writer.

Structured output

Structured output allows agents to return data in a specific, predictable format.

Instead of parsing natural language responses, you get structured data in the form of JSON objects, Pydantic models, or dataclasses that your application can use directly.

LangChain's create_agent handles structured output automatically. The user sets their desired structured output schema, and when the model generates the structured data, it's captured, validated, and returned in the 'structured_response' key of the agent's state.

Use response_format to control how the agent returns structured data:

ToolStrategy[StructuredResponseT]: Uses tool calling for structured output

ProviderStrategy[StructuredResponseT]: Uses provider-native structured output

type[StructuredResponseT]: Schema type - automatically selects best strategy based on model capabilities

None: Structured output not explicitly requested

```
def create_agent(
    ...
    response_format: Union[
        ToolStrategy[StructuredResponseT],
        ProviderStrategy[StructuredResponseT],
        type[StructuredResponseT],
        None,
    ]
```