

Python ▾

Overview

LangChain v1.0

Release notes

Migration guide

Get started

Install

Quickstart

Philosophy

Core components

Agents

Models

Messages

Tools

Short-term memory

Streaming

Advanced usage

Middleware

Guardrails

Structured output

Runtime

Context engineering

Model Context Protocol (MCP)

Human-in-the-loop

Multi-agent

Retrieval

Long-term memory

Use in production

Studio

Test

Deploy

Agent Chat UI

Observability

Get started

Quickstart

📄 Copy page

▾

💡 LangChain v1.0

Welcome to the new LangChain documentation! If you encounter any issues or have feedback, please [open an issue](#) so we can improve. Archived v0 documentation can be found [here](#).

See the [release notes](#) and [migration guide](#) for a complete list of changes and instructions on how to upgrade your code.

This quickstart takes you from a simple setup to a fully functional AI agent in just a few minutes.

Build a basic agent

Start by creating a simple agent that can answer questions and call tools. The agent will use Claude Sonnet 4.5 as its language model, a basic weather function as a tool, and a simple prompt to guide its behavior.

```
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="anthropic:claude-sonnet-4-5",
    tools=[get_weather],
    system_prompt="You are a helpful assistant",
)

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

ⓘ For this example, you will need to set up a [Claude \(Anthropic\)](#) account and get an API key. Then, set the `ANTHROPIC_API_KEY` environment variable in your terminal.

Build a real-world agent

Next, build a practical weather forecasting agent that demonstrates key production concepts:

1. **Detailed system prompts** for better agent behavior
2. **Create tools** that integrate with external data
3. **Model configuration** for consistent responses
4. **Structured output** for predictable results
5. **Conversational memory** for chat-like interactions
6. **Create and run the agent** create a fully functional agent

Let's walk through each step:

1 Define the system prompt

The system prompt defines your agent's role and behavior. Keep it specific and actionable:

```
SYSTEM_PROMPT = """You are an expert weather forecaster, who speaks in puns.
```

```
You have access to two tools:
```

- get_weather_for_location: use this to get the weather for a specific location
- get_user_location: use this to get the user's location

```
If a user asks you for the weather, make sure you know the location. If you can tell from the question that they mean wherever they are, use the get_user_location tool to find their location."""
```

2 Create tools

Tools let a model interact with external systems by calling functions you define. Tools can depend on runtime context and also interact with agent memory.

Notice below how the `get_user_location` tool uses runtime context:

```

from dataclasses import dataclass
from langchain.tools import tool, ToolRuntime

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

@dataclass
class Context:
    """Custom runtime context schema."""
    user_id: str

@tool
def get_user_location(runtime: ToolRuntime[Context]) -> str:
    """Retrieve user information based on user ID."""
    user_id = runtime.context.user_id
    return "Florida" if user_id == "1" else "SF"

```

💡 Tools should be well-documented: their name, description, and argument names become part of the model's prompt. LangChain's [@tool decorator](#) adds metadata and enables runtime injection via the `ToolRuntime` parameter.

3 Configure your model

Set up your [language model](#) with the right [parameters](#) for your use case:

```

from langchain.chat_models import init_chat_model

model = init_chat_model(
    "anthropic:claude-sonnet-4-5",
    temperature=0.5,
    timeout=10,
    max_tokens=1000
)

```

4 Define response format

Optionally, define a structured response format if you need the agent responses to match a specific schema.

```
from dataclasses import dataclass


# We use a dataclass here, but Pydantic models are also supported
@dataclass
class ResponseFormat:
    """Response schema for the agent."""
    # A punny response (always required)
    punny_response: str
    # Any interesting information about the weather if available
    weather_conditions: str | None = None
```

5 Add memory

Add [memory](#) to your agent to maintain state across interactions. This allows the agent to remember previous conversations and context.

```
from langgraph.checkpoint.memory import InMemorySaver

checkpointer = InMemorySaver()
```

 In production, use a persistent checkpointer that saves to a database. See [Add and manage memory](#) for more details.

6 Create and run the agent

Now assemble your agent with all the components and run it!

```

agent = create_agent(
    model=model,
    system_prompt=SYSTEM_PROMPT,
    tools=[get_user_location, get_weather_for_location],
    context_schema=Context,
    response_format=ResponseFormat,
    checkpoint=checkpointer
)

# `thread_id` is a unique identifier for a given conversation
config = {"configurable": {"thread_id": "1"}}

response = agent.invoke(
    {"messages": [{"role": "user", "content": "what is the we"}],
    config=config,
    context=Context(user_id="1")
)

print(response['structured_response'])
# ResponseFormat(
#     punny_response="Florida is still having a 'sun-derful'
#     weather_conditions="It's always sunny in Florida!"
# )

# Note that we can continue the conversation using the same `
response = agent.invoke(
    {"messages": [{"role": "user", "content": "thank you!"}]},
    config=config,
    context=Context(user_id="1")
)

print(response['structured_response'])
# ResponseFormat(
#     punny_response="You're 'thund-erfully' welcome! It's al
#     weather_conditions=None
# )

```

Show Full example code

Congratulations! You now have an AI agent that can:

Understand context and remember conversations

Use multiple tools intelligently

Provide structured responses in a consistent format

Handle user-specific information through context

Maintain conversation state across interactions

[Edit the source of this page on GitHub](#)



Resources

- Forum
- Changelog
- LangChain Academy
- Trust Center

Company

- About
- Careers
- Blog