Building an AI Agent from Scratch in Python

Using No Frameworks

Leonie Monigatti

What's the best way to get started building AI agent systems? There are countless frameworks for building AI agents available, such as CrewAI, LangGraph, and the OpenAI Agents SDK, and it can be overwhelming to choose one. On the other hand, Anthropic recommended starting with using direct LLM APIs calls to understand the fundamentals before relying on framework abstractions.

This tutorial takes this approach by exploring how to implement an AI agent from scratch in Python using an LLM API directly to gain a better understanding of what's happening under the hood. This tutorial focuses on implementing a singleagent before advancing to more complex topics, such as agentic workflows or multi-agent systems.

Implementing an AI Agent from scratch

This section implements an `Agent()` lass by incorporating each of the following core components of an AI agent step-by-step:

1. **LLM and instructions:** The LLM powering the agent's reasoning and decision-making capabilities with explicit guidelines defining how the agent should behave.
2. **Memory:** Conversation history (short-term memory) the agent uses to understand the current interaction.
3. **Tools:** External functions or APIs the agent can call.

And finally, we will put everything together in a loop.

Component 1: LLM and Instructions

At the core of every AI agent, you have a Large Language Model (LLM) with tool use capabilities, such as Anthropic's Claude 4 Sonnet, OpenAI's GPT-4o, or Google's Gemini 2.5 Pro.

This tutorial uses Claude 4 Sonnet through the Anthropic API but you can easily adjust the code to any other LLM API of your choice.

To use the Anthropic API, you will need an `ANTHROPIC_API_KE` , which you can obtain by creating an Anthropic account and navigating to the "API Keys" tab in your dashboard. Once you have your API key, you need to store it in the environment variables, an .env file, or the Google Colab secrets, depending on the environment you're using.
Let's install and import the required libraries.

```
%%capture
%pip install -U anthropic python-dotenv
```

```
import anthropic
import os
from dotenv import load_dotenv
from google.colab import userdata

load_dotenv()

print(anthropic.__version__)
```

```
0.69.0
```

Now, we will implement a simple `Agent` class with the following components:

- Initialization: Sets up the LLM client and configures the model with a system prompt that contains instruction for the agent on how to act. (You could also turn this into a parameter you can pass to the agent but we will use a fixed one for simplicity.)
- `chat` method: Processes user messages by sending them to the LLM API and returning the response

```
class Agent:
    """A simple AI agent that can answer questions"""

    def __init__(self):
        self.client = anthropic.Anthropic(api_key=os.getenv("ANTHROPIC_API_KEY"))
        self.model = "claude-sonnet-4-20250514"
        self.system_message = "You are a helpful assistant that breaks down problems into s

    def chat(self, message):
        """Process a user message and return a response"""

        response = self.client.messages.create(
            model=self.model,
            max_tokens=1024,
            system=self.system_message,
            messages=[
                {"role": "user", "content": message}
                ],
            temperature=0.1,
        )

        return response
```

The agent now has simple query-response capabilities. Let's test it.

```
agent = Agent()

response = agent.chat("I have 4 apples. How many do you have?")
print(response.content[0].text)
```

```
I don't have any apples - as an AI, I don't have a physical form, so I can't possess
physical objects like apples. Only you have apples in this scenario (4 of them).

Is there something you'd like to do with this information, like a math problem involving
your apples?
```

Great. Let's follow up with a second message.

```
response = agent.chat("I ate 1 apple. How many are left?")
print(response.content[0].text)
```

```
I don't have enough information to answer how many apples are left. To solve this, I would
need to know:

**What I need:**
- How many apples you started with

**The calculation would be:**
Starting number of apples - 1 apple eaten = Apples remaining

Could you tell me how many apples you had before eating one?
```

As you can see, the agent lacks the information from the first message. That's why we need to give the agent access to the conversation history.

Component 2: (Conversation) Memory

Memory in agents can take many different forms, such as short-term and long-term memory, and memory management can become a complex topic. For the sake of this tutorial, let's keep it simple and start with a basic short-term memory implementation.

Short-term memory gives the agent access to the conversation history to understand the current interaction. In its simplest form, the short-term memory is just a list of pas `message` between the `user` and the `assistant`. (Note, that the longer the conversation history becomes, you will run into context window limitations and will need to implement a more sophisticated solution.)

We implement short-term memory by adding a `messages` property where we store both:

- the user inputs v `{"role": "user", "content": mes`
- the response v `{"role": "assistant", "content": response.cont`

```
class Agent:
    """A simple AI agent that can answer questions in a multi-turn conversation"""

    def __init__(self):
        self.client = anthropic.Anthropic(api_key=os.getenv("ANTHROPIC_API_KEY"))
        self.model = "claude-sonnet-4-20250514"
        self.system_message = "You are a helpful assistant that breaks down problems into s
        self.messages = []

    def chat(self, message):
        """Process a user message and return a response"""

        # Store user input in short-term memory
        self.messages.append({"role": "user", "content": message})
```

```
        response = self.client.messages.create(
            model=self.model,
            max_tokens=1024,
            system=self.system_message,
            messages=self.messages,
            temperature=0.1,
        )

        #Store assistant's response in short-term memory
        self.messages.append({"role": "assistant", "content": response.content})

        return response
```

Now, let's test the agent again with the previous example conversation.

```
agent = Agent()

response = agent.chat("I have 4 apples. How many do you have?")
print(response.content[0].text)

response = agent.chat("I ate 1 apple. How many are left?")
print(response.content[0].text)
```

```
I don't have any apples - as an AI, I don't have a physical form and can't possess
physical objects like apples. You have 4 apples, and I have 0 apples.

Is there something you'd like to do with your 4 apples, like a math problem or recipe
suggestion?
Let me solve this step by step:

**Step 1:** Identify the starting amount
- You started with 4 apples

**Step 2:** Identify what was consumed
- You ate 1 apple

**Step 3:** Calculate the remaining amount
- Apples left = Starting amount - Apples eaten
- Apples left = 4 - 1 = 3

**Answer:** You have 3 apples left.
```

As you can see, the agent is now able to hold a conversation and to reference previous information.

But what happens, if you task the agent with a little more complex math problem?

```
agent = Agent()

response = agent.chat("What is 157.09 * 493.89?")

print(response.content[0].text)
```

```
I'll solve this step by step using the standard multiplication algorithm.

157.09 × 493.89
```

```
First, let me multiply 157.09 by each digit of 493.89:

**Step 1:** 157.09 × 9 (ones place)
157.09 × 9 = 1,413.81

**Step 2:** 157.09 × 80 (tens place)
157.09 × 8 = 1,256.72
1,256.72 × 10 = 12,567.2

**Step 3:** 157.09 × 300 (hundreds place)
157.09 × 3 = 471.27
471.27 × 100 = 47,127

**Step 4:** 157.09 × 90,000 (ten-thousands place)
157.09 × 9 = 1,413.81
1,413.81 × 10,000 = 14,138,100

**Step 5:** 157.09 × 400,000 (hundred-thousands place)
157.09 × 4 = 628.36
628.36 × 100,000 = 62,836,000

**Step 6:** Add all partial products:
```
```
      1,413.81
     12,567.2
     47,127
14,138,100
62,836,000
-----------
77,035,208.01
```

Therefore, **157.09 × 493.89 = 77,035.2081**
```

The agent's answer sounds perfectly believable but if you validate it, you can actually see that even powerful LLMs like Claude 4 Sonnet can still make arithmetic errors without tools.

```
157.09 * 493.89
```

```
77585.1801
```

Component 3: Tool Use

To extend the agent's capabilities, you can provide it with tools that can range from simple functions to using external APIs. For this tutorial, we will implement a simple `CalculatorTool` class, that can handle math problems.

The exact implemention of tool use is different across providers, but at the core always requires two key components:

- **Function implementation:** This is the actual function that executes the tool's logic, such as performing a calculation, or making an API call.
- **Tool schema:** A structured description of the tool. The tool description is important because it tells the LLM what the tool does, when to use it, and what parameters it takes.
```

This tutorial follows the Anthropic documentation on tool use. If you're using a different LLM API than this tutorial, I recommend to check out your LLM providers documentation on tool use.

```
class CalculatorTool():
    """A tool for performing mathematical calculations"""

    def get_schema(self):
        return {
            "name": "calculator",
            "description": "Performs basic mathematical calculations, use also for simple a
            "input_schema": {
                "type": "object",
                "properties": {
                    "expression": {
                        "type": "string",
                        "description": "Mathematical expression to evaluate (e.g., '2+2',
                    }
                },
                "required": ["expression"]
            }
        }

    def execute(self, expression):
        """
        Evaluate mathematical expressions.
        WARNING: This tutorial uses eval() for simplicity but it is not recommended for pro

        Args:
            expression (str): The mathematical expression to evaluate
        Returns:
            float: The result of the evaluation
        """
        try:
            result = eval(expression)
            return {"result": result}
        except:
            return {"error": "Invalid mathematical expression"}
```

Note, that in this tutorial, we are just implementing a single tool. In production code, you'd typically use an abstract base class to ensure a consistent interface across tools.

Let's test if the calculator function works.

```
calculator_tool = CalculatorTool()

calculator_tool.execute("157.09 * 493.89")
```

```
{'result': 77585.1801}
```

Now that we have a `CalculatorTool`, let's add tool use capabilities to our agent, in three steps:

1. Add `tools` and `tool_map` attributes to store available tools
2. Add the private `_get_tool_schemas()` method to extract tool schemas
3. Add tool handling logic to the `create` method to detect tool use

```
class Agent:
    """A simple AI agent that can use tools to answer questions in a multi-turn conversatio

    def __init__(self, tools):
        self.client = anthropic.Anthropic(api_key=os.getenv("ANTHROPIC_API_KEY"))
        self.model = "claude-sonnet-4-20250514"
        self.system_message = "You are a helpful assistant that breaks down problems into s
        self.messages = []
        self.tools = tools
        self.tool_map = {tool.get_schema()["name"]: tool for tool in tools}

    def _get_tool_schemas(self):
        """Get tool schemas for all registered tools"""
        return [tool.get_schema() for tool in self.tools]

    def chat(self, message):
        """Process a user message and return a response"""

        # Store user input in short-term memory
        self.messages.append({"role": "user", "content": message})

        response = self.client.messages.create(
            model=self.model,
            max_tokens=1024,
            system=self.system_message,
            tools=self._get_tool_schemas() if self.tools else None,
            messages=self.messages,
            temperature=0.1,
        )

        # Store assistant's response in short-term memory
        self.messages.append({"role": "assistant", "content": response.content})

        return response
```

Let’s give it a try.

```
calculator_tool = CalculatorTool()
agent = Agent(tools=[calculator_tool])

response = agent.chat("What is 157.09 * 493.89?")

for block in response:
    print(block)
```

```
('id', 'msg_01BzC2FerKEr8rC1wGfaMiNK')
('content', [TextBlock(citations=None, text="I'll calculate 157.09 * 493.89 for you.",
type='text'), ToolUseBlock(id='toolu_017NhVhd5wYWdEw7fFRPHyXL', input={'expression':
'157.09 * 493.89'}, name='calculator', type='tool_use')])
('model', 'claude-sonnet-4-20250514')
('role', 'assistant')
('stop_reason', 'tool_use')
('stop_sequence', None)
('type', 'message')
('usage', Usage(cache_creation=CacheCreation(ephemeral_1h_input_tokens=0,
```

```
ephemeral_5m_input_tokens=0), cache_creation_input_tokens=0, cache_read_input_tokens=0,
input_tokens=433, output_tokens=77, server_tool_use=None, service_tier='standard'))
```

As you can see in the response, the agent answers with “I’ll calculate 157.09 * 493.89 for you.” but instead of calculating the expression itself, it stops with `stop_reason` being `ol_use` is means, that the agent is waiting for the user to execute the tool and return the result from the tool to the agent.

But now, the agent has responded that it needs help with executing the tool and is waiting. This is where the final component of the loop comes into play.

Component 4: Agent Loop

You might have already heard people say that “Agents are models using tools in a loop”. Without the loop, the agent can only handle single-turn without multi-turn interactions.

I really like this pseudo code by Barry Zhan, Anthropic, showing that agents are just LLMs making decisions in a loop, observing results, and deciding what to do next.

```
env = Environment()
tools = Tools(env)
system_prompt = "Goals, constraints, and how to act"

while True:
 action = llm.run(system_prompt + env.state)
 env.state = tools.run(action)
```

For this simple agent implementation, that means, we have the following flow:

1. User sends message to agent
2. Agent decides it needs a tool and responds with a `stop_reason` of `ool_use` nd a `ol_use` ck with the tool name and parameters. It’s saying “I’m pausing for you to execute this tool with these parameters”.
3. The user executes the tool and sends the tool result back to the agent in a follow-up message
4. The agent continues and gives the final response.

```
import json

def run_agent(user_input, max_turns=10):
 calculator_tool = CalculatorTool()
 agent = Agent(tools=[calculator_tool])

 i = 0

   while i < max_turns: # It's safer to use max_turns rather than while True
     i += 1
     print(f"\nIteration {i}:")

     print(f"User input: {user_input}")
     response = agent.chat(user_input)
     print(f"Agent output: {response.content[0].text}")

     # Handle tool use if present
     if response.stop_reason == "tool_use":

          # Process all tool uses in the response
```

```
        tool_results = []
        for content_block in response.content:
            if content_block.type == "tool_use":
                tool_name = content_block.name
                tool_input = content_block.input

                print(f"Using tool {tool_name} with input {tool_input}")

                #Execute the tool
                tool = agent.tool_map[tool_name]
                tool_result = tool.execute(**tool_input)

                tool_results.append({
                    "type": "tool_result",
                    "tool_use_id": content_block.id,
                    "content": json.dumps(tool_result)
                })
                print(f"Tool result: {tool_result}")

        #Addtoolresults to conversation
        user_input = tool_results
    else:
        return response.content[0].text

return
```

Testing the Implemented AI Agent

Let's test the implemented AI agent with a few example test cases.

Test 1: General question (no tool use)

This test demonstrates the agent's ability to answer a simple, general question that does not require the use of any external tools.

```
response = run_agent("I have 4 apples. How many do you have?")
```

```
Iteration 1:
User input: I have 4 apples. How many do you have?
Agent output: I don't have any apples since I'm an AI assistant - I don't have a physical
form or possessions. But I can help you with calculations involving your 4 apples if you
need!

Is there something specific you'd like to calculate or figure out with your 4 apples?
```

Test 2: Tool Use

This test demonstrates how the agent understands that it needs to use a tool to to solve a specific task and uses the `CalculatorTool` get the correct result.

```
response = run_agent("What is 157.09 * 493.89?")
```

```
Iteration 1:
User input: What is 157.09 * 493.89?
Agent output: I'll calculate 157.09 * 493.89 for you.
Using tool calculator with input {'expression': '157.09 * 493.89'}
Tool result: {'result': 77585.1801}

Iteration 2:
User input: [{'type': 'tool_result', 'tool_use_id': 'toolu_01FC9yLWt2Cf6a8zLGhj7ZJz',
'content': '{"result": 77585.1801}'}]
Agent output: The result of 157.09 * 493.89 is **77,585.1801**.
```

Test 3: Step-by-step tool use

This test demonstratesthe agent's ability to break down a more complex problem into smaller steps and use the `CalculatorTool` ultiple times within a single conversation to arrive at the final answer.

```
response = run_agent("If my brother is 32 years younger than my mother and my mother is 30
```

```
Iteration 1:
User input: If my brother is 32 years younger than my mother and my mother is 30 years
older than me and I am 20, how old is my brother?
Agent output: I'll solve this step by step using the given information.

Given:
- You are 20 years old
- Your mother is 30 years older than you
- Your brother is 32 years younger than your mother

Let me calculate your mother's age first:
Using tool calculator with input {'expression': '20 + 30'}
Tool result: {'result': 50}

Iteration 2:
User input: [{'type': 'tool_result', 'tool_use_id': 'toolu_01WPMQRzCi4roua9vQ7qXeCR',
'content': '{"result": 50}'}]
Agent output: So your mother is 50 years old.

Now I'll calculate your brother's age:
Using tool calculator with input {'expression': '50 - 32'}
Tool result: {'result': 18}

Iteration 3:
User input: [{'type': 'tool_result', 'tool_use_id': 'toolu_01UL7n7a85XJUn7Tgk8kiHhX',
'content': '{"result": 18}'}]
Agent output: Your brother is 18 years old.

To summarize:
- You: 20 years old
- Your mother: 50 years old (30 years older than you)
- Your brother: 18 years old (32 years younger than your mother)
```

Summary

This tutorial showed you how you can implement a minimal AI agent from scratch using just an LLM API without any frameworks. Hopefully, you now understand the fundamentals of what happens under the hood of an AI agent and what people mean, when they say “Agents are models using tools in a loop”.

You can find this notebook in this GitHub repository

As a next step, you can refer to the following resources to learn more about how to implement different agent workflows.

Resources

- Anthropic’s Building Effective Agents Cookbook
- Build an AI Agent from SCRATCH with Python! (No Frameworks) by Aaron Dunn
- Building Effective LLM Workflows in Pure Python by Dave Ebbelaar

Hi, I am Leonie, a machine learning engineer and technical writer. I help developers build vector-based AI solutions. My writing focuses on machine learning and AI engineering.