LEARNINGS

Build Your First AI Agent

Learn how to transform a Large Language Model into an AI agent that can reason, use tools, and complete tasks. This practical guide walks you through building a simple runnable agent in Python.

AI LLM AI AGENTS PYTHON

Ibtissam Oulahchoum

13 September 2026

Large Language Models are incredibly good at understanding and generating language. You can ask them questions, give them instructions, summari

But by themselves, LLMs are limited.

They cannot automatically interact with your database, call your APIs, search for information, or perform actions inside your application.

An AI agent changes that.

In this article, we'll build a simple mental model for understanding how AI agents work and create a small runnable AI agent in Python.

What Is an AI Agent?

At its simplest, an AI agent is a system that uses an LLM to understand a goal, decide what to do next, and use available tools to complete a task.

A traditional LLM interaction might look like this:

```
User → LLM → Response
```

An AI agent can do more:

```
User → LLM → Decision → Tool → Result → LLM → Response
```

The important difference is the ability to take actions.

The LLM acts as the decision-making layer, while your application provides the tools and environment where actions can happen.

The Four Core Components

A basic AI agent usually consists of four main parts.

1. The LLM

The LLM is responsible for understanding the user's request and deciding what should happen.

For example, imagine a user asks:

> *What is the square root of 14,567 and round it to two decimal places?*

Instead of trying to solve the problem manually, the model can decide to use a calculator.

The important idea is this:

> *The LLM does not need to perform every action itself. It can decide which action should be performed.*

2. Instructions

Instructions define how the agent should behave.

They give the LLM context about its role, responsibilities, and limitations.

For example:

```
You are a helpful AI assistant.

Your goal is to answer user questions accurately.

When a calculation is required, use the calculator tool.

Never invent tool results.
```

Good instructions make an agent more predictable.

3. Tools

Tools give an agent capabilities beyond generating text.

A tool could be:

- A calculator
- A database
- A search engine
- An API
- A file system
- An email service

A tool is usually just a function that your application makes available to the agent.

For example:

```
def calculate(expression):
    return eval(expression)
```

However, in production applications, you should avoid using eval() with arbitrary user input. We will use a safer implementation in the runnable examp

4. The Execution Loop

The execution loop connects the LLM and the tools.

The process looks like this:

1. User sends a request
2. LLM understands the request
3. LLM decides whether a tool is needed
4. Application executes the tool
5. Tool result is sent back to the LLM
6. LLM generates the final response

This is the foundation of an AI agent.

Building a Runnable AI Agent in Python

Let's build a small agent that can answer questions and use a calculator when it needs to perform arithmetic.

Step 1: Install the Dependency

Create a new Python environment and install the SDK:

```
pip install openai
```

Then set your API key as an environment variable.

On macOS or Linux:

```
export OPENAI_API_KEY="your_api_key_here"
```

On Windows PowerShell:

```
$env:OPENAI_API_KEY="your_api_key_here"
```

Your application will read the API key from the environment rather than hardcoding it into the source code.

Step 2: Create a Calculator Tool

Our agent needs a tool that can safely perform basic mathematical operations.

```
import ast
import operator


OPERATORS = {
    ast.Add: operator.add,
    ast.Sub: operator.sub,
    ast.Mult: operator.mul,
    ast.Div: operator.truediv,
    ast.Pow: operator.pow,
    ast.USub: operator.neg,
}


def calculate(expression: str):
    """Safely evaluate a basic mathematical expression."""

    def evaluate(node):
        if isinstance(node, ast.Constant):
            return node.value

        if isinstance(node, ast.BinOp):
            operator_type = type(node.op)

            if operator_type not in OPERATORS:
                raise ValueError("Unsupported operation")

            return OPERATORS[operator_type](
                evaluate(node.left),
                evaluate(node.right),
            )

        if isinstance(node, ast.UnaryOp):
            operator_type = type(node.op)

            if operator_type not in OPERATORS:
                raise ValueError("Unsupported operation")

            return OPERATORS[operator_type](evaluate(node.operand))

        raise ValueError("Invalid expression")

    tree = ast.parse(expression, mode="eval")

    return evaluate(tree.body)
```

This function accepts basic arithmetic operations such as:

```
10 + 20
25 * 4
100 / 5
2 ** 10
```

Unlike eval(), this implementation only allows the operations we explicitly define.

Step 3: Describe the Tool to the LLM

The LLM needs to know that the calculator exists and understand when it should use it.

We can describe the tool using a function schema.

```
tools = [
    {
        "type": "function",
        "function": {
            "name": "calculate",
            "description": "Perform a mathematical calculation.",
            "parameters": {
                "type": "object",
                "properties": {
                    "expression": {
                        "type": "string",
                        "description": (
                            "A mathematical expression using "
                            "+, -, *, /, and **."
                        ),
                    }
                },
                "required": ["expression"],
                "additionalProperties": False,
            },
        },
    }
]
```

This does not execute the function.

It simply tells the LLM:

- A function called calculate is available. Use it when a mathematical calculation is required.

Step 4: Connect the LLM and the Tool

Now we can create the agent.

```
import json
from openai import OpenAI


client = OpenAI()


SYSTEM_PROMPT = """
You are a helpful AI assistant.

Answer user questions clearly.

When a mathematical calculation is required,
use the calculate tool instead of calculating
the result yourself.
"""

messages = [
    {
        "role": "system",
        "content": SYSTEM_PROMPT,
    }
]


def run_agent(user_message: str):
    messages.append(
        {
            "role": "user",
            "content": user_message,
        }
    )

    response = client.chat.completions.create(
        model="gpt-5",
        messages=messages,
        tools=tools,
    )

    message = response.choices[0].message

    messages.append(message)

    if message.tool_calls:
        for tool_call in message.tool_calls:

            if tool_call.function.name == "calculate":

                arguments = json.loads(
                    tool_call.function.arguments
                )

                result = calculate(
                    arguments["expression"]
                )

                messages.append(
                    {
                        "role": "tool",
                        "tool_call_id": tool_call.id,
                        "content": str(result),
                    }
```

```
                )

        final_response = client.chat.completions.create(
            model="gpt-5",
            messages=messages,
        )

        return final_response.choices[0].message.content

    return message.content
```

Step 5: Run the Agent

Now we can ask our agent a question.

```
answer = run_agent(
    "What is 125 multiplied by 48?"
)

print(answer)
```

The agent's internal process looks approximately like this:

```
User:
What is 125 multiplied by 48?

↓

LLM:
I should use the calculator.

↓

Tool Call:
calculate("125 * 48")

↓

Python Tool:
Returns 6000

↓

LLM:
125 multiplied by 48 is 6,000.
```

The important part is that the Python function and the LLM are working together.

The Complete Example

Here is the complete runnable implementation:

```
import ast
import json
import operator

from openai import OpenAI


client = OpenAI()


OPERATORS = {
    ast.Add: operator.add,
    ast.Sub: operator.sub,
    ast.Mult: operator.mul,
    ast.Div: operator.truediv,
    ast.Pow: operator.pow,
    ast.USub: operator.neg,
}


def calculate(expression: str):

    def evaluate(node):

        if isinstance(node, ast.Constant):
            return node.value

        if isinstance(node, ast.BinOp):
            operator_type = type(node.op)

            if operator_type not in OPERATORS:
                raise ValueError(
                    "Unsupported operation"
                )

            return OPERATORS[operator_type](
                evaluate(node.left),
                evaluate(node.right),
            )

        if isinstance(node, ast.UnaryOp):
            operator_type = type(node.op)

            if operator_type not in OPERATORS:
                raise ValueError(
                    "Unsupported operation"
                )

            return OPERATORS[operator_type](
                evaluate(node.operand)
            )

        raise ValueError("Invalid expression")

    tree = ast.parse(
        expression,
        mode="eval",
    )

    return evaluate(tree.body)
```

```
tools = [
    {
        "type": "function",
        "function": {
            "name": "calculate",
            "description": (
                "Perform a mathematical calculation."
            ),
            "parameters": {
                "type": "object",
                "properties": {
                    "expression": {
                        "type": "string",
                        "description": (
                            "A mathematical expression."
                        ),
                    }
                },
                "required": [
                    "expression"
                ],
                "additionalProperties": False,
            },
        },
    }
]


SYSTEM_PROMPT = """
You are a helpful AI assistant.

Answer questions clearly.

Use the calculate tool whenever
a mathematical calculation is required.
"""


messages = [
    {
        "role": "system",
        "content": SYSTEM_PROMPT,
    }
]


def run_agent(user_message: str):

    messages.append(
        {
            "role": "user",
            "content": user_message,
        }
    )

    response = client.chat.completions.create(
        model="gpt-5",
        messages=messages,
        tools=tools,
    )

    assistant_message = (
        response.choices[0].message
```

```
    )

    messages.append(
        assistant_message
    )

    if assistant_message.tool_calls:

        for tool_call in (
            assistant_message.tool_calls
        ):

            if (
                tool_call.function.name
                == "calculate"
            ):

                arguments = json.loads(
                    tool_call.function.arguments
                )

                result = calculate(
                    arguments["expression"]
                )

                messages.append(
                    {
                        "role": "tool",
                        "tool_call_id": (
                            tool_call.id
                        ),
                        "content": str(result),
                    }
                )

        final_response = (
            client.chat.completions.create(
                model="gpt-5",
                messages=messages,
            )
        )

        return (
            final_response
            .choices[0]
            .message
            .content
        )

    return assistant_message.content


answer = run_agent(
    "What is 125 multiplied by 48?"
)

print(answer)
```

What Is Actually Happening?

The most important thing to understand is that the LLM is not directly running your Python code.

The flow is:

1. LLM
2. Requests a tool call
3. Your Python application receives the request
4. Your application executes the function
5. The result is returned to the LLM
6. The LLM generates the final answer

This separation gives you control over what your agent is allowed to do.

You decide:

- Which tools exist
- What each tool can access
- Which actions are allowed
- How tool results are validated

The Agent Loop

Our first implementation only performs one tool call before returning a response.

More advanced agents use a loop:

1. Understand
2. Decide
3. Call a Tool
4. Observe the Result
5. Need Another Action?
6. Yes → Repeat
7. No → Respond

This allows an agent to solve more complex tasks involving multiple steps and multiple tools.

Where to Go Next

Once you understand this simple implementation, you can gradually add more capabilities:

- Multiple tools
- API integrations
- Database access
- File processing
- Retrieval-Augmented Generation
- Conversation memory
- Planning
- Multi-step execution

But the core idea remains the same.

> *The LLM decides what action may be needed. Your application controls how that action is executed.*

Key Takeaway

An AI agent is not just an LLM.

It is a system built around an LLM.

The LLM provides understanding and decision-making, while tools give the system the ability to interact with the outside world.

A simple agent can be built with:

> ***An LLM + Instructions + Tools + an Execution Loop***

Start with one tool and one clear task.

Once you understand the connection between the model, your application, and external tools, you have the foundation needed to build more capable