

Chapter 1

From an LLM to an Agent

Agentic Engineering: AI Agents from First Principles

Central question

What actually changes when a large language model becomes an agent?

By the end of this chapter, you will have built the smallest useful agent in the book.

1.1 A model can answer. An agent can act.

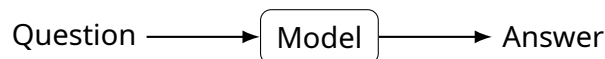
Suppose we ask a language model a simple question: “What is 37×42 ?” The model receives text and returns text. It may answer correctly. It may answer incorrectly. Either way, nothing outside the model has changed. No calculator has been called, no file has been read, no database has been queried, and no observation from the world has returned to influence a second decision.

That distinction is the starting point for this book. A large language model is a powerful generator and decision component, but generation alone is not agency. Agency appears when a model is embedded in software that can translate a decision into an action, observe the consequence, and use that observation to decide what to do next.

```
answer = model.generate([
    {"role": "user", "content": "What is 37 * 42?"}
])

print(answer)
```

This is useful software, but its architecture is still one-way:



The model can describe an action—“I should use a calculator”—without actually causing one. Giving a model instructions about tools is not the same thing as connecting those instructions to executable capabilities.

Experiment 1 — A model call

Our first experiment is deliberately tiny. We use a deterministic stand-in for a model so that every reader sees exactly the same output. The stand-in returns an imperfect arithmetic answer. This is not intended to measure the arithmetic ability of any current model. It isolates the architecture.

```
from agentic.model import ScriptedModel

model = ScriptedModel([
    "37 x 42 is approximately 1,500."
])

messages = [
    {"role": "user", "content": "What is 37 * 42?"}
]

answer = model.generate(messages)

print("MODEL:", answer)
print("GROUND TRUTH:", 37 * 42)
print("CORRECT:", str(37 * 42) in answer)
```

Running it produces:

```
MODEL: 37 x 42 is approximately 1,500.
GROUND TRUTH: 1554
```

CORRECT: False

What did this experiment establish?

Only that the model produced a response. It did not establish that models are generally bad at arithmetic, nor that an agent is required for multiplication. The arithmetic task gives us a ground truth and a capability we can add in the next experiment.

1.2 Add an action

Now give the software a calculator. The model no longer needs to be the component that performs the arithmetic. Its job can change from answering the mathematical question directly to choosing an appropriate action.

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

tools = {
    "calculator": calculator
}
```

In production code we would not use unrestricted `eval`. Our runnable repository uses a restricted arithmetic evaluator. Here the short version keeps attention on the architecture.

The model can now propose a structured decision:

```
{
  "action": "calculator",
  "arguments": {
    "expression": "37*42"
  }
}
```

But even now the model has not performed the action. Ordinary software receives that proposal, looks up the capability, executes it, and returns the result.

```
decision = model.generate(messages)

tool = tools[decision["action"]]
result = tool(**decision["arguments"])
```

The value 1554 is an observation produced by the environment, not a fact invented by the model.

Experiment 2 — From model to agent

```
from agentic.model import ScriptedModel
from agentic.tools import calculator

model = ScriptedModel([
    {
```

```

        "action": "calculator",
        "arguments": {"expression": "37*42"}
    },
    {
        "done": True,
        "answer": "37 x 42 = 1554."
    },
]

messages = [
    {"role": "user", "content": "What is 37 * 42?"}
]

decision = model.generate(messages)
result = calculator(**decision["arguments"])

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

final = model.generate(messages)

print("ACTION:", decision["action"])
print("OBSERVATION:", result)
print("ANSWER:", final["answer"])

```

The deterministic run produces:

```

ACTION: calculator
OBSERVATION: 1554
ANSWER: 37 x 42 = 1554.

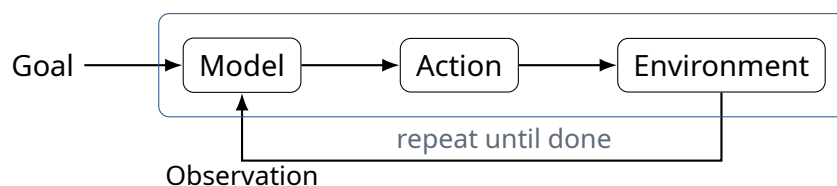
```

1.3 The important line is not the calculator

It is tempting to say that the second program is an agent because it has a tool. That is close, but it misses the most important mechanism. A tool is only a capability. The architectural transition is the feedback path:

Decision → Action → Observation → Next decision

The result of the action becomes information available to the model's next decision. Once that feedback path can repeat, we have the core of an agent loop.



This is why a chatbot that merely describes actions is different from an agent that can perform them. The difference is not personality, verbosity, or whether the prompt contains the word *agent*. The difference is executable coupling to an environment.

1.4 A first definition

We can now introduce the working model that will organize the rest of the book:

Agent = Model + Context + Actions + State + Control

Model decides. **Context** is what the model can see now. **Actions** are what the system can do. **State** preserves information across steps. **Control** determines what may happen, when another step occurs, and when execution stops.

At this point those five terms are intentionally broad. We will engineer each one separately. Context will grow into retrieval and context engineering. State will grow into memory and checkpoints. Actions will grow into tools, protocols, coding, and computer use. Control will grow into ReAct, planning, routing, evaluation, permissions, budgets, and human approval.

Component	In this chapter	Later in the book
Model	Chooses an answer or action	Model routing, critics, judges
Context	Messages supplied to the model	RAG, compression, context engineering
Actions	Calculator	Tools, MCP, coding, computer use
State	Tool result appended to messages	Memory, shared state, checkpoints
Control	One decision followed by another	Loops, planning, budgets, permissions

1.5 The model is not the whole system

This leads to one of the book's central themes: agent performance is a property of a system, not only a model. Two applications can use the same underlying model and behave very differently because one gives it better context, safer actions, stronger validation, more useful feedback, or better stopping rules.

We will eventually summarize that idea as:

$$\text{Agent quality} \approx \text{Model quality} + \text{Harness quality}$$

The symbol is conceptual, not a numerical equation. The point is that the surrounding software—the harness—can improve or destroy the usefulness of the model. A stronger model does not remove the need for engineering around it.

1.6 Deliberately break it

Every important mechanism in this book will be broken on purpose. Failure makes the boundary of an abstraction visible.

Break Experiment 2 by removing the calculator while leaving the model free to request it. The model can still produce:

```
{
  "action": "calculator",
  "arguments": {"expression": "37*42"}
}
```

but the software cannot execute the request. The model’s decision may be perfectly sensible while the system still fails. This distinction—reasoning success versus execution success—will return throughout the book.

1.7 What should we measure?

For these deterministic demonstrations, the measurements are simple:

Measure	Model only	Tool-using agent
Task success	No in our scripted run	Yes
Model calls	1	2
Tool calls	0	1

These are demonstration results from deterministic scripts, not empirical claims about current frontier models. Later experiments will use repeated live-model trials and add tokens, cost, latency, reliability, and risk.

1.8 The smallest useful agent

We can now strip the architecture down to its essence. Ignore provider SDKs, tracing systems, databases, frameworks, and deployment infrastructure. The conceptual machine is:

```
messages = [goal]

while not done:
    decision = model(messages)

    if decision.done:
        return decision.answer

    observation = execute(decision)
    messages.append(observation)
```

That loop is intentionally incomplete. It has no validation, no step limit, no error handling, no permissions, and no cost budget. Those omissions are not mistakes in the pedagogy. They are the agenda for the rest of the book.

Chapter principle

Agency begins when model decisions can cause actions whose consequences become observations for subsequent decisions.

1.9 Before we add more

It would be easy to keep adding tools and call the result increasingly agentic. We will not do that yet. The next chapter addresses a more basic problem: the model communicates in probabilistic natural language, while software needs dependable contracts.

Before an agent can safely control tools, we need to answer:

How can probabilistic language control deterministic software?

Exercises

1. Change Experiment 1 so that the scripted model returns the correct answer. Does that make it an agent? Explain why or why not.
2. Add a second tool called `square`. What new problem appears when the model must choose between capabilities?
3. Remove the tool observation from `messages` before the second model call. What information has the system lost?
4. Replace the calculator with a function that returns the current time. Which component of `Model + Context + Actions + State + Control` changed?
5. Write one example of a system that uses an LLM but should remain a deterministic workflow rather than an agent. We will revisit your answer in Chapter 4.

Chapter 1 checklist

- ☐ An LLM call is not, by itself, an agent.
- ☐ A tool is a capability; software still executes the capability.
- ☐ The critical transition is Decision → Action → Observation → Next decision.
- ☐ Agent = Model + Context + Actions + State + Control.
- ☐ System quality depends on both the model and its surrounding harness.

Repository

Runnable files for this chapter:

```
01_llm_call.py  
02_llm_to_agent.py  
agentic/model.py  
agentic/tools.py
```

The repository uses a deterministic `ScriptedModel` in the first examples so that the mechanism and expected output remain stable. A live-provider adapter can be introduced separately without changing the architecture.