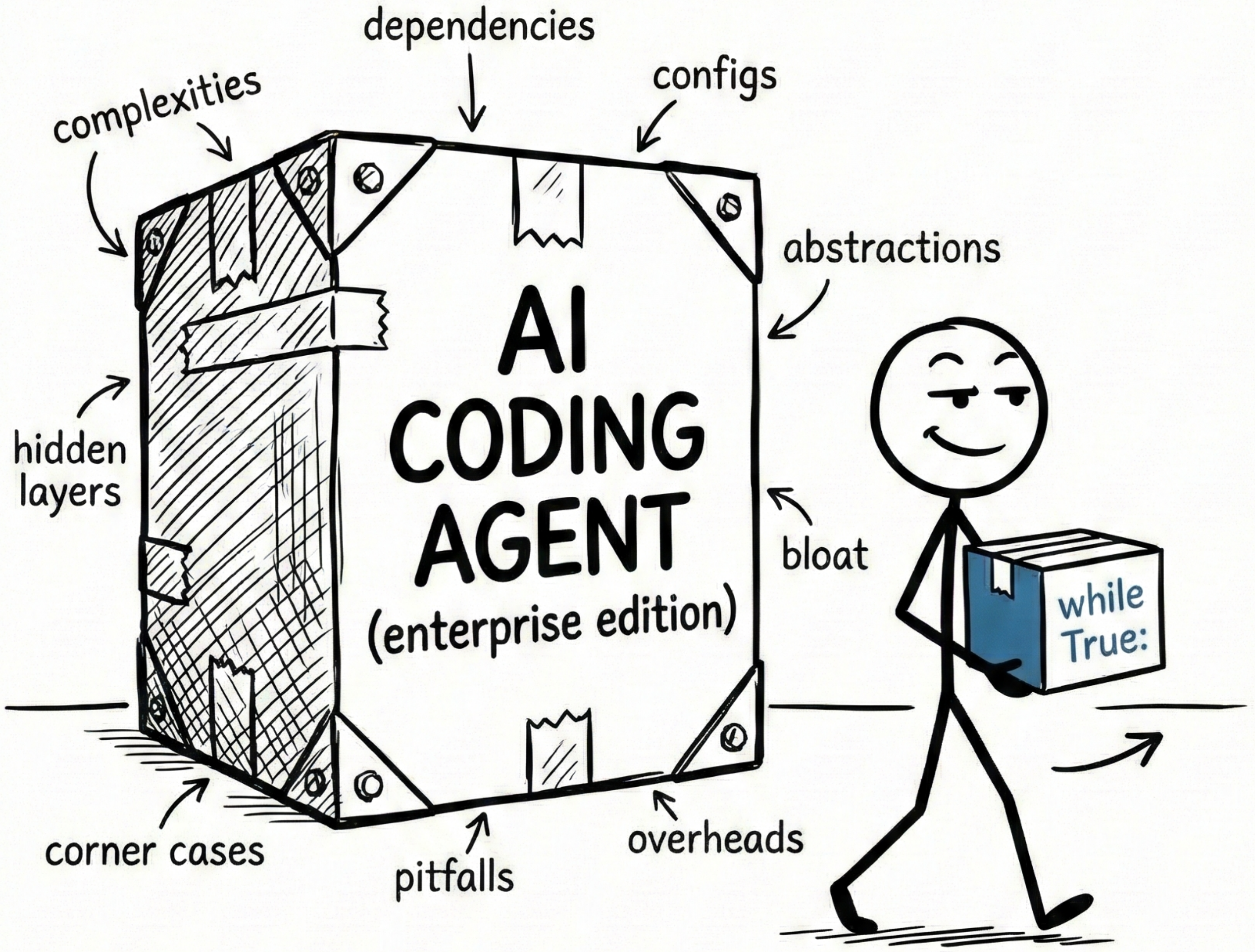
Build Your Own CODING AGENT

The Zero-Magic Guide to AI Agents in Pure Python



Owen Ou

Build Your Own Coding Agent

The Zero-Magic Guide to AI Agents in Pure Python

Owen Ou

This book is available at
https://leanpub.com/build-your-own-coding-agent

This version was published on 2026-02-07

Tweet This Book!

Please help Owen Ou by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Just built a coding agent from scratch with nothing but HTTP calls and a shell. Turns out ”AI magic” is just a while loop and an API call.

The suggested hashtag for this book is #buildyourowncodingagent.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#buildyourowncodingagent

To my wife, my son, my parents, and my grandparents—you taught me there is no magic, only effort. Everything I build, I build because of you.

Contents

Preface 1

Who This Book Is For 1

What You Will Build 1

Testing Approach 1

Code Examples 2

Conventions Used in This Book 2

Chapter 1: The Zero Magic Manifesto 4

What is an Agent, Really? 4

What We Are Building 5

Project Setup 6

The AgentStop Exception 8

The Agent Class 8

Defining Success with Tests 9

The Main Loop 11

Run It 11

Wrapping Up 12

Chapter 2: The Raw Request 13

Step 1: Get an API Key 13

Step 2: The Vault (.env) 13

Step 3: The Anatomy of a Request 14

Step 4: The Code 15

Step 5: Run It 16

Troubleshooting 17

Cleaning Up 17

Wrapping Up 18

Chapter 3: The Infinite Loop 19

The Illusion of Memory 19

The Testing Problem 19

Response Types . . . 20
The FakeBrain Pattern . . . 21
Defining Success . . . 22
The Claude Class . . . 24
The Agent Class (Updated) . . . 25
The Main Loop (Updated) . . . 27
Verify the Tests Pass . . . 27
Test the Memory . . . 28
The Context Window Problem . . . 28
Wrapping Up . . . 29

Chapter 4: The Universal Adapter . . . 30
The Adapter Pattern . . . 30
HTTP Resilience . . . 30
The Brain Interface . . . 30
The FakeBrain (Updated) . . . 30
The Claude Brain (Refactored) . . . 30
The DeepSeek Brain . . . 30
The BRAINS Registry . . . 31
The Agent Class (Updated) . . . 31
Tests for Multi-Brain Support . . . 31
The Main Loop (Updated) . . . 31
Setting Up DeepSeek . . . 31
Try It . . . 31
Why This Matters . . . 31
Wrapping Up . . . 32

Chapter 5: The Tool Protocol . . . 33
How Tools Actually Work . . . 33
Defining the Tool Interface . . . 33
The ReadFile Tool . . . 33
The WriteFile Tool . . . 33
Tool Helpers . . . 33
Updating the Thought Class . . . 33
Updating the Claude Class . . . 34
The Agent Class with Tools . . . 34
The Main Loop . . . 34
Test It . . . 34
Wrapping Up . . . 34

Chapter 6: The Scratchpad (Memory) **35**
The "Zero Magic" Memory 35
The Memory Class 35
The ToolContext Class 35
The SaveMemory Tool 35
Updating the Claude Class 35
Crafting the System Prompt 35
Updating the Agent Class 36
The Main Loop (Updated) 36
Testing Persistence 36
Why This is Powerful 36
Wrapping Up 36

Chapter 7: The Safety Harness (Planner Mode) **37**
The Concept 37
Tests First 37
Extending ToolContext 37
The Guarded WriteFile Tool 37
The Agent Class (Updated) 37
The Main Loop (Updated) 37
Testing the Harness 38
The Psychology of the "Plan" 38
Wrapping Up 38

Chapter 8: The Context Pipeline (Map & Search) **39**
The ListFiles Tool 39
The SearchCodebase Tool 39
Update the Tools List 39
The "Zoom In" Test 39
Wait, is this RAG? 39
The Architectural Significance 39
Wrapping Up 40

Chapter 9: The Reality Check (Running Code) **41**
The Feedback Loop 41
Tests First 41
The RunCommand Tool 41
The Interactive Trap 41
The Self-Healing Demo 41

The TDD Workflow 41
The Surgical Edit 42
Why This Changes Everything 42
Security Considerations 42
Wrapping Up 42

Chapter 10: Going Dark (Local Models) 43
The Trade-off 43
Installing Ollama 43
The Ollama Brain Class 43
Running with Ollama 43
The "Infinite Loop" Experiment 43
The Practical Differences 43
The Hybrid Workflow 44
Model Selection 44
Troubleshooting Ollama 44
Wrapping Up 44

Chapter 11: The Extension (Web Search) 45
Step 1: The Meta-Prompt 45
Step 2: The Surgery 45
Step 3: The Reference Implementation 45
Step 4: The Test 45
Why This Matters 45
Wrapping Up 45

Chapter 12: The Capstone (Building a Game) 47
Step 1: Preparation 47
Step 2: The Architect (Plan Mode) 48
Step 3: The Builder (Act Mode) 48
Step 4: The Reality Check 49
Step 5: The Pivot (Feature Creep) 50
The Result 50
Wrapping Up 51
Epilogue: The End of the Beginning 51

Acknowledgments 53

Preface

Who This Book Is For

You are a software engineer who is skeptical of AI hype.

You have seen the demos. You have tried the frameworks. You have watched your LangChain app hallucinate its way into deleting a production database. And you thought: "*There has to be a better way.*"

There is. This book is for developers who want to understand what is actually happening when an AI agent runs. Not the marketing diagrams. Not the "Reasoning Engine" abstractions. The actual HTTP requests. The actual `while` loop.

If you can read Python and have built a web app or CLI tool before, you have everything you need.

What You Will Build

Nanocode is a coding agent that runs in your terminal. By the end of this book, it will:

- Read and write files in your codebase
- Execute shell commands
- Search code using pure Python
- Remember context across sessions
- Ask for permission before dangerous operations
- Search the web for documentation and answers

You will build it from scratch using only `requests`, `subprocess`, and `pytest`. No LangChain. No vector databases. No "orchestration frameworks." Just Python you can debug with `print()`.

The one exception: Chapter 11 adds `ddgs` for web search—a single lightweight dependency.

Testing Approach

This book uses a *test-alongside* approach. For each feature:

1. We introduce the concept (why we need this)
2. We show the test first (what success looks like)
3. We implement the code (make the test pass)
4. We verify with pytest (prove it works)

This teaches Test-Driven Development thinking without tedious red-green-refactor cycles in print. More importantly, it solves a critical problem: you cannot test an LLM-powered application by actually calling the LLM. API calls are slow, expensive, and non-deterministic.

Starting in Chapter 3, we introduce the `FakeBrain` pattern—a test double that returns predictable responses. This lets you run your entire test suite without making a single API call or spending a single cent.

Code Examples

This book follows a “Code First” approach. Each chapter builds on the previous one, and every code example is extracted from working files.

To get the code:

- **GitHub:** Clone or download from GitHub.[1]
- **Leanpub:** The complete source code is also included in the downloadable resources with your purchase.

The code is organized by chapter (`ch01/`, `ch02/`, etc.). Each folder contains:

- `nanocode.py` – The complete, runnable agent for that chapter
- `test_nanocode.py` – Tests that verify the code works without API calls

You can copy any chapter folder and pick up from there. Run `pytest` to verify your code matches the expected behavior.

[1]https://github.com/owenthereal/build-your-own-coding-agent

Conventions Used in This Book

Throughout the book, you will see three types of callouts:

Dev Tip: Architectural wisdom that applies beyond this project. These are patterns you can reuse in your own work.

Warning: Security or safety risks. Pay attention to these–ignoring them can lead to deleted files or leaked API keys.

Aside: Deep dives and tangents. Useful context, but you can skip them on a first read without losing the thread.

Code walkthroughs follow a consistent pattern: first the context (why we need this code), then the code itself, then a line-by-line explanation of the important parts.

Let's build.

Chapter 1: The Zero Magic Manifesto

If you have tried to build an AI application in the last two years, you have likely felt *Framework Fatigue*.

You install a popular library. You import a `ReasoningEngine`. You call `.run()`. It works like magic for the "Hello World" example. But the moment you try to do something real—like editing a specific line in a Python file without deleting the imports—it breaks.

And because you used a framework, you can't fix it. You are stuck digging through layers of abstract classes, factory patterns, and "Chains" trying to find the one prompt that is causing the hallucination.

We are not going to do that here.

This book is a rebellion against "Magic." We are going to build a production-grade coding agent called **Nanocode**. We will build it in pure Python. We will not use LangChain, AutoGPT, or Pydantic.

Why? Because an autonomous agent is not magic. It is just a `while` loop.

What is an Agent, Really?

Strip away the venture capital marketing, and an "Agent" is just a **thermostat**.

A thermostat reads the temperature (input), compares it to the target (decision), and turns on the heater (action). Then it waits and repeats. That's it. An AI agent does the same thing, just with text instead of temperature.

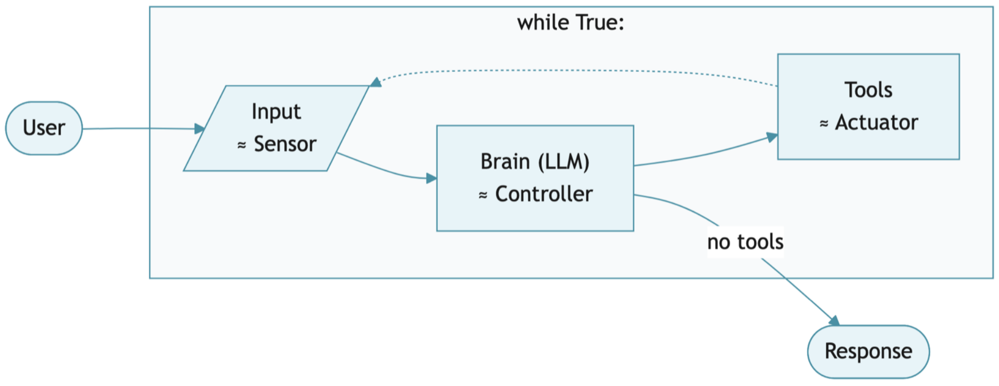


Figure 1. The Agent Loop: User input flows through a while loop where Input (Sensor) feeds the Brain/LLM (Controller), which triggers Tools (Actuator), cycling back to Input until the Brain outputs a Response.

More specifically, an agent is composed of four parts:

1. **The Brain:** The LLM (Claude, DeepSeek). A stateless function. You send text; it returns text.
2. **The Tools:** Functions the Brain can “call” (Read File, Run Command).
3. **The Memory:** The conversation history. A Python list.
4. **The Loop:** A `while True` that cycles through the above until the task is done.

Note: This list exists only in memory–it dies when the program ends. We’ll add persistent storage in Chapter 6.

If you can write a `while` loop, you can build an agent.

By building it from scratch, you will have something the framework users don’t: **Control**. When our agent gets stuck in a loop, you will know exactly which line of code caused it. When the API bill gets too high, you will see exactly where the tokens are leaking.

What We Are Building

Nanocode is a CLI tool that runs in your terminal. You talk to it like a colleague. It reads your files. It runs your commands. It edits your code.

By the end of this book, you will have built:

- **The Brain:** Connects to Claude Sonnet 4.5 (or DeepSeek, or a local model via Ollama).
- **The Hands:** Tools to read files, write files, and run shell commands.
- **The Eyes:** A search tool that finds code using `git grep`.
- **The Safety Harness:** A "Planner Mode" that prevents accidental `rm -rf /`.

But first, we set up the workshop.

Project Setup

Dev Tip: The biggest threat to an AI project is "Dependency Rot." AI libraries move fast and break things. By sticking to `requests` and `subprocess`, this agent will likely still run 5 years from now.

1. Initialize the Project

```
mkdir nanocode
cd nanocode
git init
```

2. Create a Virtual Environment

Never install AI tools globally. They conflict with system packages.

```
# Mac/Linux
python3 -m venv venv
source venv/bin/activate

# Windows
python -m venv venv
venv\Scripts\activate
```

3. Install Dependencies

We only need three libraries:

- `requests` – To talk to the LLM APIs.
- `python-dotenv` – To load API keys from a `.env` file.
- `pytest` – To test our code without making API calls.

Create `requirements.txt`:

```
requests
python-dotenv
pytest
```

Install:

```
pip install -r requirements.txt
```

4. Secure Your Keys

Warning: If you push your API key to GitHub, bots will scrape it and drain your account within minutes.

Create `.gitignore`:

```
.env
__pycache__/
venv/
.DS_Store
.nanocode/
```

The AgentStop Exception

Before we write the event loop, we need a clean way to exit. Instead of using `break` statements scattered throughout the code, we'll define an exception that signals "the agent should stop."

The Context: Exceptions are not just for errors; they are also a control flow mechanism. When the user types `/q`, we raise `AgentStop`. The main loop catches it and exits cleanly.

The Code:

```python
# --- Exceptions ---

class AgentStop(Exception):
    """Raised when the agent should stop processing."""
    pass
```

This goes at the top of `nanocode.py`, right after the imports. It's a marker exception—no logic, just a signal.

The Agent Class

Now the core abstraction: the `Agent` class. This encapsulates the agent's state and behavior in a testable unit.

The Context: We *could* put all the logic in `main()`. But then we'd have to mock `input()` and `print()` to test it. By extracting the logic into `Agent.handle_input()`, we can test it directly.

The Code:

```python
10  class Agent:
11      """A coding agent that processes user input."""
12  
13      def __init__(self):
14          pass
15  
16      def handle_input(self, user_input):
17          """Handle user input. Returns output string, raises AgentStop to quit."""
18          if user_input.strip() == "/q":
19              raise AgentStop()
20  
21          if not user_input.strip():
22              return ""
23  
24          return f"You said: {user_input}\n(Agent not yet connected)"
```

The Walkthrough:

- **Lines 13-14:** Empty constructor for now. We'll add `brain` and `tools` in later chapters.
- **Lines 18-19:** Check for the `/q` quit command. Raise `AgentStop` instead of returning a special value.
- **Lines 21-22:** Skip empty input. Return empty string (no output to display).
- **Line 24:** Echo the input back. This is a placeholder—later, we'll send this to the Brain.

Defining Success with Tests

Before we write the main loop, let's define what success looks like. Tests serve as executable documentation.

Create `test_nanocode.py`:

```python
import pytest
from nanocode import Agent, AgentStop


def test_handle_input_returns_string():
    """Verify handle_input returns a string for normal input."""
    agent = Agent()
    result = agent.handle_input("hello")
    assert isinstance(result, str)
    assert "hello" in result


def test_empty_input_returns_empty_string():
    """Verify empty/whitespace input returns empty string."""
    agent = Agent()
    assert agent.handle_input("") == ""
    assert agent.handle_input("   ") == ""
    assert agent.handle_input("\n") == ""


def test_quit_command_raises_agent_stop():
    """Verify /q raises AgentStop exception."""
    agent = Agent()
    with pytest.raises(AgentStop):
        agent.handle_input("/q")


def test_quit_command_with_whitespace():
    """Verify /q works with surrounding whitespace."""
    agent = Agent()
    with pytest.raises(AgentStop):
        agent.handle_input("  /q  ")
```

Run the tests:

```
pytest test_nanocode.py -v
```

```
test_nanocode.py::test_handle_input_returns_string PASSED
test_nanocode.py::test_empty_input_returns_empty_string PASSED
test_nanocode.py::test_quit_command_raises_agent_stop PASSED
test_nanocode.py::test_quit_command_with_whitespace PASSED
```

All green. Our agent handles the basic cases correctly.

Aside: Why pytest? It discovers functions starting with `test_` and runs them. No boilerplate, no classes required. The test code itself is plain Python—no magic.

The Main Loop

Now the thin I/O wrapper that connects the agent to the terminal:

```
29  def main():
30      agent = Agent()
31      print("⚡ Nanocode v0.1 initialized.")
32      print("Type '/q' to quit.")
33
34      while True:
35          try:
36              user_input = input("\n❯ ")
37              output = agent.handle_input(user_input)
38              if output:
39                  print(output)
40
41          except (AgentStop, KeyboardInterrupt):
42              print("\nExiting...")
43              break
44
45
46  if __name__ == "__main__":
47      main()
```

The Walkthrough:

- **Lines 30-32:** Create the agent and print startup messages.
- **Line 36:** `input()` blocks and waits for the user to type something.
- **Lines 37-39:** Call `handle_input()` and print any output.
- **Lines 41-43:** Catch `AgentStop` (from `/q`) or `KeyboardInterrupt` (from Ctrl+C) and exit cleanly.

Aside: Python's `input()` reads one line at a time. All prompts in this book are single-line. This keeps the code simple—production agents use richer input methods like readline or full TUIs.

Notice the separation: `Agent.handle_input()` contains all the logic. `main()` is just I/O glue. This makes the agent testable without mocking stdin/stdout.

Run It

```
python nanocode.py
```

You should see:

```
⚡ Nanocode v0.1 initialized.
Type '/q' to quit.

❯ hello
You said: hello
(Agent not yet connected)

❯ /q

Exiting...
```

This is the chassis. The engine comes next.

Wrapping Up

In this chapter, you built the foundation of an AI agent: an `Agent` class with a `handle_input()` method, controlled by a simple event loop. More importantly, you embraced the *Zero Magic* philosophy—no frameworks, no abstractions, just raw Python you can understand and control.

You learned that an agent is composed of four parts: the Brain (LLM), the Tools (functions), the Memory (conversation history), and the Loop (the `while True` that ties it all together). If you can write a `while` loop, you can build an agent.

You also wrote tests before verifying the behavior manually. This test-first approach will pay dividends as the codebase grows—each chapter builds on the last, and tests ensure we don't break what already works.

In the next chapter, we'll wake up the brain by connecting to the Claude API using nothing but the `requests` library. No SDK magic—just raw HTTP.

Chapter 2: The Raw Request

In Chapter 1, we built the body. Now we need to wake up the brain.

Most tutorials will tell you to `pip install anthropic`. We are not going to do that.

Why? Because SDKs hide the truth. They add layers of abstraction that make "Hello World" easy but make debugging "Error 400" a nightmare. When you use an SDK, you are learning a library. When you use raw HTTP, you are learning the protocol.

In this chapter, we will send a message to Claude using nothing but the `requests` library. Claude is Anthropic's flagship LLM—one of the most capable models for coding tasks.

Step 1: Get an API Key

To talk to Claude, you need an API Key. This is a long string of characters that acts as your credit card.

1. Go to the Anthropic Console.[1]
2. Sign up and add a payment method (minimum $5 credit).
3. Create a new API Key and name it `nanocode`.
4. Copy the key (it starts with `sk-ant-...`).

⚠ **Warning:** Treat this key like a password. Anyone with it can spend your money.

Step 2: The Vault (.env)

We need a safe place to store this key. We never put keys directly in code.

Create a file named `.env` in your project root:

[1]https://console.anthropic.com/settings/keys

```
touch .env
```

Open it and paste your key:

```
ANTHROPIC_API_KEY=sk-ant-api03-...
```

We installed `python-dotenv` in Chapter 1 for exactly this purpose—it reads `.env` and loads the values into `os.environ`.

Step 3: The Anatomy of a Request

To talk to an LLM, we send an HTTP POST request to:

`https://api.anthropic.com/v1/messages`

This request needs three things:

1. **Authentication (Headers):** “Here is my ID card.”
2. **Configuration (Body):** “I want to talk to model X with max Y tokens.”
3. **Message (Body):** “Hello!”

The Headers

Anthropic requires three headers:

Header	Value	Purpose
`x-api-key`	Your secret key	Authentication
`anthropic-version`	`2023-06-01`	API version
`content-type`	`application/json`	Format

The Payload

The “Messages API” expects a list of message dictionaries:

```
"messages": [
    {"role": "user", "content": "Hello, world!"}
]
```

Each message has a `role` (either `"user"` or `"assistant"`) and `content` (the text).

Step 4: The Code

Create a file called `test_api.py`. This is a "smoke test" to prove our connection works. We will delete it later.

The Context: We are writing linear, procedural code. No functions, no classes. We want to see the bare metal.

The Code:

```python
import os
import requests
import json
from dotenv import load_dotenv

# 1. Load the vault
load_dotenv()
api_key = os.getenv("ANTHROPIC_API_KEY")

# Basic check so we don't crash with a confusing "NoneType" error later
if not api_key:
    print("Error: ANTHROPIC_API_KEY not found in .env")
    exit(1)

# 2. Define the target
url = "https://api.anthropic.com/v1/messages"

# 3. Authenticate
headers = {
    "x-api-key": api_key,
    "anthropic-version": "2023-06-01",
    "content-type": "application/json"
}

# 4. Construct the payload
payload = {
    "model": "claude-sonnet-4-5-20250929",
    "max_tokens": 4096,
    "messages": [
30          {"role": "user", "content": "Hello, are you ready to code?"}
31      ]
32  }
33
34  # 5. Fire! (No safety net)
35  print("📡 Sending request to Claude...")
36  response = requests.post(url, headers=headers, json=payload)
37
38  # 6. Inspect the raw result
39  print(f"Status: {response.status_code}")
40
41  if response.status_code == 200:
42      # Success: Print the beautiful JSON
43      print("Response:")
44      print(json.dumps(response.json(), indent=2))
45  else:
46      # Failure: Print the ugly raw text so we can debug
47      print("Error:", response.text)
```

The Walkthrough:

- **Line 7:** `load_dotenv()` finds the `.env` file and loads variables into `os.environ`.
- **Line 8:** We grab the API key. Never hardcode this.
- **Lines 11-13:** Basic sanity check. Without this, a missing key causes a confusing `NoneType` error in the headers dict.
- **Line 21:** The `anthropic-version` header is mandatory. Omit it, and the API rejects you.
- **Line 27:** `claude-sonnet-4-5-20250929` specifies which model we want.
- **Line 28:** `max_tokens` is required. It caps the response length and prevents runaway costs.
- **Line 36:** We fire the request. No try/except—if the network is down, let Python crash. You need to see where it fails.
- **Lines 41-47:** Check the status code. 200 means success (pretty-print the JSON). Anything else means we print the raw error text for debugging.

Step 5: Run It

```
python test_api.py
```

If everything works, you should see:

```
Status: 200
Response:
{
  "id": "msg_01...",
  "type": "message",
  "role": "assistant",
  "content": [
    {
      "type": "text",
      "text": "Hello! Yes, I'm ready to code..."
    }
  ],
  "stop_reason": "end_turn",
  "usage": {
    "input_tokens": 15,
    "output_tokens": 81
  }
}
```

Troubleshooting

Error	**Cause**	**Fix**
401 Unauthorized	Bad API key	Check `.env` is loading. Print `os.environ.get("ANTHROPIC_-API_KEY")` to verify.
400 Bad Request	Malformed JSON	Did you forget `max_tokens`? Is `messages` a list?
429 Rate Limit	Too many requests or no credits	Wait, or add credits to your account.

Cleaning Up

We proved we can talk to the brain. Delete `test_api.py`—we will integrate this logic into `nanocode.py` properly in the next chapter.

Note: This `test_api.py` is throwaway code—a one-time smoke test. Real automated tests (with `FakeBrain` and pytest) come in Chapter 3. You should always delete this file after verifying your API connection works.

Dev Tip: Every API call costs money. Input tokens (what you send) are cheap. Output tokens (what the model writes) are ~5x more expensive. Keep your prompts concise.

Aside: To monitor your spending, check the Usage tab in the Anthropic Console. A typical coding session with 20-30 exchanges costs $0.10-$0.50 with Claude Sonnet. The `usage` object in each response (lines 169-172 above) shows exact token counts—you could log these to track costs programmatically.

Wrapping Up

In this chapter, you made your first direct connection to an LLM—no SDK, no magic, just a raw HTTP POST request. You learned the anatomy of an API call: headers for authentication, a payload with your message, and JSON parsing to extract the response.

You also learned why we avoid SDKs: they hide the truth. When you work with raw HTTP, you understand the protocol. When something breaks, you know exactly where to look.

In the next chapter, we'll give Claude memory by implementing the *context loop*—the technique that makes LLMs appear to remember previous messages.

Chapter 3: The Infinite Loop

We have a problem.

Run the script from Chapter 2 twice. Say "My name is Alice." Claude says hello. Run it again and ask "What is my name?" Claude says "I don't know."

This is because LLMs are *stateless*. They have total amnesia. Every request is the first time you have ever met.

To build an agent, we need to fix this. We need to create artificial memory.

The Illusion of Memory

"Memory" in an LLM isn't a hard drive. It's a log file.

When you chat with ChatGPT, it doesn't "remember" what you said 5 minutes ago. Behind the scenes, the code sends the **entire conversation history** back to the model with every new message.

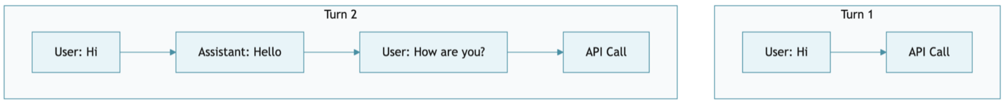


Figure 2. Context accumulation: Turn 1 sends just "User: Hi" to the API. Turn 2 sends the full history–"User: Hi", "Assistant: Hello", "User: How are you?"–to the API.

The model sees the full transcript every time. That's the trick.

We are going to implement this *context loop* manually. But first, we need to make our code testable.

The Testing Problem

Here's a hard truth: you cannot test an LLM-powered application by actually calling the LLM.

API calls are slow (2-10 seconds each), expensive (real money per call), and non-deterministic (you might get a different response every time). Imagine running a test suite that costs $5 and takes 20 minutes. You'd never run it.

The solution is *dependency injection*. Instead of hardcoding the API call inside our agent, we pass in a "brain" object. In production, the brain is Claude. In tests, the brain is a fake that returns predictable responses.

This pattern is so fundamental that we'll establish it now, before writing any more production code.

Response Types

Before we build the brain, we need to define what it returns. Claude's API sends back complex JSON with multiple content blocks. We need simple Python objects to work with.

The Context: Claude can return text, tool calls, or both in a single response. We need data classes to represent these possibilities.

The Code:

```python
class ToolCall:
    """A tool invocation request from the brain."""

    def __init__(self, id, name, args):
        self.id = id
        self.name = name
        self.args = args  # dict
```

`ToolCall` represents the brain asking us to execute a tool. The `id` is a unique identifier for tracking (Claude needs it when we report results back). The `name` is which tool to run. The `args` is a dictionary of parameters.

We won't use `ToolCall` in this chapter—the brain can't call tools yet—but we define it now because it's part of the `Thought` response type. In Chapter 5, you'll see how Claude returns these when it wants to read a file or execute a command.

```python
class Thought:
    """Standardized response from any Brain."""

    def __init__(self, text=None, tool_calls=None):
        self.text = text  # str or None
        self.tool_calls = tool_calls or []  # list of ToolCall
```

A `Thought` is what the brain returns after thinking. It might have text, tool calls, both, or neither. This abstraction will let us swap Claude for DeepSeek later without changing any other code.

The FakeBrain Pattern

Now we can build a fake brain for testing.

The Context: We need a brain that returns predictable responses, tracks how many times it was called, and records what conversation it received.

The Code:

```python
class FakeBrain:
    """Fake brain for testing - returns predictable responses."""

    def __init__(self, responses=None):
        self.responses = responses or [Thought(text="Fake response")]
        self.call_count = 0
        self.last_conversation = None

    def think(self, conversation):
        self.last_conversation = list(conversation)  # Store a copy
        if self.call_count < len(self.responses):
            response = self.responses[self.call_count]
            self.call_count += 1
            return response
        return Thought(text="No more responses")
```

This goes in `test_nanocode.py`, not the production code. The key insight: `FakeBrain` has the same interface as our real brain will—a `think()` method that takes a conversation and returns a `Thought`.

Dive Deeper: This pattern–replacing a real dependency with a predictable fake for testing–is called *dependency injection*. Martin Fowler's article "Mocks Aren't Stubs"[1] explains the variations (fakes, stubs, mocks, spies). For LLM testing, a simple fake with canned responses is usually all you need.

Defining Success

Before writing the production code, let's define what success looks like. These tests will guide our implementation.

Test 1: The brain returns a response

```python
def test_handle_input_returns_brain_response():
    """Verify handle_input returns the brain's response text."""
    brain = FakeBrain(responses=[Thought(text="Hello from brain!")])
    agent = Agent(brain=brain)
    result = agent.handle_input("hi")
    assert result == "Hello from brain!"
```

Notice we pass `brain=brain` to the Agent. This is dependency injection in action.

Test 2: Conversation accumulates

```python
def test_conversation_accumulates():
    """Verify conversation list grows with each interaction."""
    brain = FakeBrain(responses=[
        Thought(text="Response 1"),
        Thought(text="Response 2")
    ])
    agent = Agent(brain=brain)

    agent.handle_input("First message")
    assert len(agent.conversation) == 2  # user + assistant

    agent.handle_input("Second message")
    assert len(agent.conversation) == 4  # 2 users + 2 assistants
```

[1]https://martinfowler.com/articles/mocksArentStubs.html

This is the memory test. After each exchange, the conversation should have both the user message and the assistant response.

Test 3: Correct message structure

```python
def test_conversation_contains_correct_roles():
    """Verify conversation has correct role alternation."""
    brain = FakeBrain(responses=[Thought(text="AI response")])
    agent = Agent(brain=brain)

    agent.handle_input("User message")

    assert agent.conversation[0]["role"] == "user"
    assert agent.conversation[0]["content"] == "User message"
    assert agent.conversation[1]["role"] == "assistant"
    assert agent.conversation[1]["content"] == "AI response"
```

The messages must have the exact format Claude expects: `{"role": "user", "content": "..."}`.

Test 4: Brain receives the conversation

```python
def test_brain_receives_conversation():
    """Verify brain.think is called with the conversation list."""
    brain = FakeBrain()
    agent = Agent(brain=brain)

    agent.handle_input("Test message")

    assert brain.last_conversation is not None
    assert len(brain.last_conversation) == 1
    assert brain.last_conversation[0]["content"] == "Test message"
```

The brain must receive the full conversation, not just the current message.

Run these tests now—they should all fail:

```
pytest test_nanocode.py -v
```

```
FAILED test_nanocode.py::test_handle_input_returns_brain_response
FAILED test_nanocode.py::test_conversation_accumulates
...
```

Good. Now let's make them pass.

The Claude Class

Time to build the real brain.

The Context: We need a class that wraps the Claude API. It should handle authentication, send conversation history, and parse the response into a `Thought`.

The Code:

```python
class Claude:
    """Claude API - the brain of our agent."""

    def __init__(self):
        self.api_key = os.getenv("ANTHROPIC_API_KEY")
        if not self.api_key:
            raise ValueError("ANTHROPIC_API_KEY not found in .env")
        self.model = "claude-sonnet-4-5-20250929"
        self.url = "https://api.anthropic.com/v1/messages"

    def think(self, conversation):
        headers = {
            "x-api-key": self.api_key,
            "anthropic-version": "2023-06-01",
            "content-type": "application/json"
        }
        payload = {
            "model": self.model,
            "max_tokens": 4096,
            "messages": conversation
        }

        print("(Claude is thinking...)")
        response = requests.post(self.url, headers=headers, json=payload,
        ↪  timeout=120)
        response.raise_for_status()
        return self._parse_response(response.json()["content"])
```

The Walkthrough:

- **Lines 39-42:** Load the API key and fail fast if it's missing.
- **Lines 43-44:** Store config. We'll make the model configurable later.
- **Line 46:** The `think()` method is the brain's interface—same as `FakeBrain`.
- **Lines 52-55:** The payload includes `"messages": conversation`—the full history, not just the current message. This is the context loop.
- **Line 61:** Parse Claude's complex response format into our simple `Thought`.

Now the response parser:

```
    def _parse_response(self, content):
        """Convert Claude's response format to Thought."""
        text_parts = []
        tool_calls = []

        for block in content:
            if block["type"] == "text":
                text_parts.append(block["text"])
            elif block["type"] == "tool_use":
                tool_calls.append(ToolCall(
                    id=block["id"],
                    name=block["name"],
                    args=block["input"]
                ))

        return Thought(
            text="\n".join(text_parts) if text_parts else None,
            tool_calls=tool_calls
        )
```

Claude's API returns a list of "content blocks." Each block has a `type`—either `"text"` or `"tool_use"`. We collect all text blocks into a single string and convert tool_use blocks into `ToolCall` objects.

The Agent Class (Updated)

Now we update the `Agent` from Chapter 1 to accept a brain and maintain conversation history.

The Code:

```
 86  class Agent:
 87      """A coding agent with conversation memory."""
 88  
 89      def __init__(self, brain):
 90          self.brain = brain
 91          self.conversation = []
 92  
 93      def handle_input(self, user_input):
 94          """Handle user input. Returns output string, raises AgentStop to
         ↪   quit."""
 95          if user_input.strip() == "/q":
 96              raise AgentStop()
 97  
 98          if not user_input.strip():
 99              return ""
100  
101          self.conversation.append({"role": "user", "content": user_input})
102  
103          try:
104              thought = self.brain.think(self.conversation)
105              text = thought.text or ""
106              self.conversation.append({"role": "assistant", "content": text})
107              return text
108          except Exception as e:
109              self.conversation.pop()  # Remove failed user message
110              return f"Error: {e}"
```

The Walkthrough:

- **Lines 89-91:** Accept a brain via dependency injection. Initialize an empty conversation list.
- **Line 101:** Append the user's message to history *before* calling the brain.
- **Lines 103-107:** Call the brain, extract the text, append the response to history.
- **Lines 108-110:** If the API call fails, remove the user message we just added. This keeps the conversation in a valid state.

The key insight is line 101: we add the user message *before* calling the brain. The brain needs to see the full conversation including the current message.

Dev Tip: When things go wrong, your first debugging tool is `print(self.conversation)`. This shows exactly what the brain is seeing.

> Malformed messages, missing roles, or truncated content become obvious when you inspect the raw list.

The Main Loop (Updated)

The main loop is now just a thin I/O wrapper:

```python
def main():
    brain = Claude()
    agent = Agent(brain)
    print("⚡ Nanocode v0.2 (Memory Active)")
    print("Type '/q' to quit.\n")

    while True:
        try:
            user_input = input("❯ ")
            output = agent.handle_input(user_input)
            if output:
                print(f"\n{output}\n")

        except (AgentStop, KeyboardInterrupt):
            print("\nExiting...")
            break


if __name__ == "__main__":
    main()
```

All the logic is in the `Agent` class. The loop just reads input, calls `handle_input()`, and prints the result. This separation makes the agent testable—we test `Agent.handle_input()` directly without needing to mock `input()` or `print()`.

Verify the Tests Pass

Run the tests again:

```
pytest test_nanocode.py -v
```

```
test_nanocode.py::test_handle_input_returns_brain_response PASSED
test_nanocode.py::test_conversation_accumulates PASSED
test_nanocode.py::test_conversation_contains_correct_roles PASSED
test_nanocode.py::test_brain_receives_conversation PASSED
```

All green. The tests verify our implementation without making a single API call.

Test the Memory

Now test with the real brain:

```
python nanocode.py
```

Try this conversation:

```
❯ I am building a Python agent.
(Claude is thinking...)

That sounds exciting! Building a Python agent is a great project...

❯ What language am I using?
(Claude is thinking...)

You are using Python.
```

It remembers. We have built a stateful chatbot.

The Context Window Problem

You might be thinking: “Can I keep this running forever?”

No.

Every loop iteration, the `messages` list grows:

Turn	Approximate Tokens
1	50
10	5,000
100	50,000

Eventually, you hit the *context limit* (200k tokens for Claude Sonnet). Exceed it, and the API returns `400 Bad Request`.

Aside: Real agents solve this by "pruning"—deleting old messages from the beginning of the list, or summarizing them. For our prototype, this is fine—restart the agent to clear the history.

Dev Tip: Notice `print("(Claude is thinking...)")`? Network calls take 2-10 seconds. Always give immediate feedback that the Enter key worked, or users will hit Ctrl+C thinking it froze.

Wrapping Up

In this chapter, you solved the *stateless problem*. LLMs have total amnesia—every request is a fresh start. The trick is to send the entire conversation history with every message, creating the illusion of memory.

You built a *context loop*: a growing list of messages that accumulates user input and assistant responses. This is how ChatGPT and every other chatbot "remembers" your conversation.

More importantly, you learned the *FakeBrain pattern*—dependency injection that lets you test your agent without API calls. This pattern will appear throughout the book. Every brain we build (Claude, DeepSeek, Ollama) will have the same `think()` interface, and `FakeBrain` will test them all.

In the next chapter, we'll clean up our code using the *Adapter Pattern*, making it trivial to switch between Claude, DeepSeek, or any other LLM provider.

Chapter 4: The Universal Adapter

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

The Adapter Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

HTTP Resilience

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

The Brain Interface

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

The FakeBrain (Updated)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

The Claude Brain (Refactored)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

The DeepSeek Brain

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

The BRAINS Registry

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

The Agent Class (Updated)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

Tests for Multi-Brain Support

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

The Main Loop (Updated)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

Setting Up DeepSeek

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

Try It

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

Why This Matters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

Wrapping Up

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

Chapter 5: The Tool Protocol

How Tools Actually Work

Defining the Tool Interface

The ReadFile Tool

The WriteFile Tool

Tool Helpers

Updating the Thought Class

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

Updating the Claude Class

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

The Agent Class with Tools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

The Main Loop

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

Test It

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

Wrapping Up

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

Chapter 6: The Scratchpad (Memory)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

The “Zero Magic” Memory

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

The Memory Class

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

The ToolContext Class

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

The SaveMemory Tool

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

Updating the Claude Class

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

Crafting the System Prompt

Updating the Agent Class

The Main Loop (Updated)

Testing Persistence

Why This is Powerful

Wrapping Up

Chapter 7: The Safety Harness (Planner Mode)

The Concept

Tests First

Extending ToolContext

The Guarded WriteFile Tool

The Agent Class (Updated)

The Main Loop (Updated)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

Testing the Harness

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

The Psychology of the "Plan"

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

Wrapping Up

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

Chapter 8: The Context Pipeline (Map & Search)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

The ListFiles Tool

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

The SearchCodebase Tool

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

Update the Tools List

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

The “Zoom In” Test

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

Wait, is this RAG?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

The Architectural Significance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

Wrapping Up

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

Chapter 9: The Reality Check (Running Code)

The Feedback Loop

Tests First

The RunCommand Tool

The Interactive Trap

The Self-Healing Demo

The TDD Workflow

The Surgical Edit

Why This Changes Everything

Security Considerations

Wrapping Up

Chapter 10: Going Dark (Local Models)

The Trade-off

Installing Ollama

The Ollama Brain Class

Running with Ollama

The “Infinite Loop” Experiment

The Practical Differences

The Hybrid Workflow

Model Selection

Troubleshooting Ollama

Wrapping Up

Chapter 11: The Extension (Web Search)

Step 1: The Meta-Prompt

Step 2: The Surgery

Step 3: The Reference Implementation

Step 4: The Test

Why This Matters

Wrapping Up

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/build-your-own-coding-agent.

Chapter 12: The Capstone (Building a Game)

We have reached the end of the road.

We started with a minimal script that could barely echo "Hello." We ended with Nanocode v1.0: an autonomous agent with:

- **Brains:** Support for Claude, DeepSeek, and local models via Ollama
- **Hands:** Tools to read, write, and edit files
- **Eyes:** Tools to map and search the codebase
- **Memory:** A persistent scratchpad
- **Safety:** A plan mode harness
- **Curiosity:** A web search tool to learn new things

Now, we must answer the only question that matters: *Can it actually code?*

In this final chapter, we perform a "Zero Code Challenge." We will build a classic Snake game using Python and Pygame. The rule: you are not allowed to write a single line of Python. You can only speak to the agent in English.

Aside: This demo involves many API calls. If you hit rate limits (HTTP 429), the agent will retry automatically. For long sessions, consider using a local model via Ollama to avoid limits entirely.

Step 1: Preparation

Create a working directory for the game and navigate into it:

```
mkdir -p snake_game
cd snake_game
```

If you are using the book's code repository, copy the agent from ch11:

```
cp ../ch11/nanocode.py .
cp ../.env .  # Your API key
```

(If you built nanocode.py from scratch, copy your latest version and `.env` file into this folder instead.)

Install Pygame:

```
pip install pygame
```

Aside: On Windows, this should work out of the box. On macOS, you may need to install SDL libraries first: `brew install sdl2 sdl2_image sdl2_mixer sdl2_ttf`. On Linux, install the SDL dev packages: `sudo apt install libsdl2-dev libsdl2-image-dev libsdl2-mixer-dev libsdl2-ttf-dev`.

Step 2: The Architect (Plan Mode)

Start the agent. We begin in plan mode because we want a blueprint before we lay bricks.

```
python nanocode.py
```

The Prompt:

```
Build a classic Snake game using Pygame. Include a score counter and Game Over
↪ screen with restart. Put ALL code in ONE file: snake.py. Write the plan in
↪ PLAN.md.
```

The agent will use `write_file` to create `PLAN.md`. Read it. It should outline the Snake class, the Food class, and the game loop—all in a single file.

If it looks good, approve it.

Step 3: The Builder (Act Mode)

Switch to act mode:

```
/mode act
```

The Prompt:

```
Implement the plan in snake.py. All code in one file.
```

Watch the terminal:

```
  → Writing snake.py
```

The agent is generating code based on the context it stored in `PLAN.md`.

Step 4: The Reality Check

Now, we run it.

The Prompt:

```
Run the game with: python snake.py
```

The agent executes `run_command`. A window pops up. You play Snake.

> **Dev Tip:** The game window blocks the agent. When you run `python snake.py`, the agent waits until you close the game window before continuing. This is normal—Pygame's main loop holds the terminal.

If it crashes: LLMs are non-deterministic. Your agent might produce a bug on the first try. If the game crashes with an error like `AttributeError: 'Snake' object has no attribute 'draw'`, don't fix it yourself. Let the agent see the stderr.

The Prompt:

```
The game crashed. Read the error and fix it.
```

The agent will read the traceback, use `read_file` to find the bug, use `edit_file` to patch it, and run again.

Dev Tip: This is the feedback loop from Chapter 9 in action. The agent writes, runs, reads the error, and fixes. You just watch.

Step 5: The Pivot (Feature Creep)

The game works, but it's ugly. The snake is just green squares. Let's stress-test the agent's ability to refactor.

The Prompt:

```
The game looks boring. Make the snake change color as it eats food, increase
↪ speed every 5 points, and search the web for 'cool retro game color
↪ palettes' to apply.
```

The agent should:

1. Use `search_web` to find color palettes
2. Use `read_file` to understand the current rendering logic
3. Use `edit_file` to inject the new features
4. Run the game to verify

This is the full toolkit in action: search, read, edit, run.

The Result

If everything worked, you now have a playable Snake game. You didn't write a single line of Python. You spoke English, and the agent translated your intent into working code.

Warning: Your results may vary. LLMs are non-deterministic. The agent might take a different approach, produce different bugs, or need more iterations. That's normal. The point isn't that it works perfectly on the first try—it's that it *converges* on a working solution through iteration.

Wrapping Up

In this chapter, you put Nanocode through its final exam: building a complete game from scratch using only English prompts.

You experienced:

1. **Plan mode:** The agent architected the solution in `PLAN.md`
2. **Act mode:** The agent implemented the code
3. **The feedback loop:** The agent debugged its own crashes
4. **The full toolkit:** read, write, edit, search, run—all working together

The key insight: an autonomous coding agent isn't magic. It's the combination of simple tools (file I/O, subprocess, web search) orchestrated by a language model. You built every piece yourself.

Epilogue: The End of the Beginning

You have just built a video game without writing code.

But you built something more important than a Snake game. You built *the machine that builds the machine.*

You now possess a tool that grants you:

- **Leverage:** Spin up prototypes in minutes, not hours
- **Fearlessness:** Dive into unknown codebases because your agent can map and explain them
- **Independence:** Run local models and keep your data private

Where to go from here?

This `nanocode.py` is yours. It's under 700 lines of Python. You understand every line because you wrote it.

Some ideas:

- **Add Git:** Write a tool to commit changes automatically
- **Add Vision:** Give Claude screenshots of your frontend so it can fix CSS
- **Add Voice:** Hook up Whisper so you can talk to your agent while walking
- **Add MCP:** Connect to external services via the Model Context Protocol

The AI revolution isn't about waiting for a god-like model to save us. It's about engineers like you building the harnesses that make these models useful.

You are no longer just a coder. You are an AI engineer.

Now, go build something impossible.

Acknowledgments