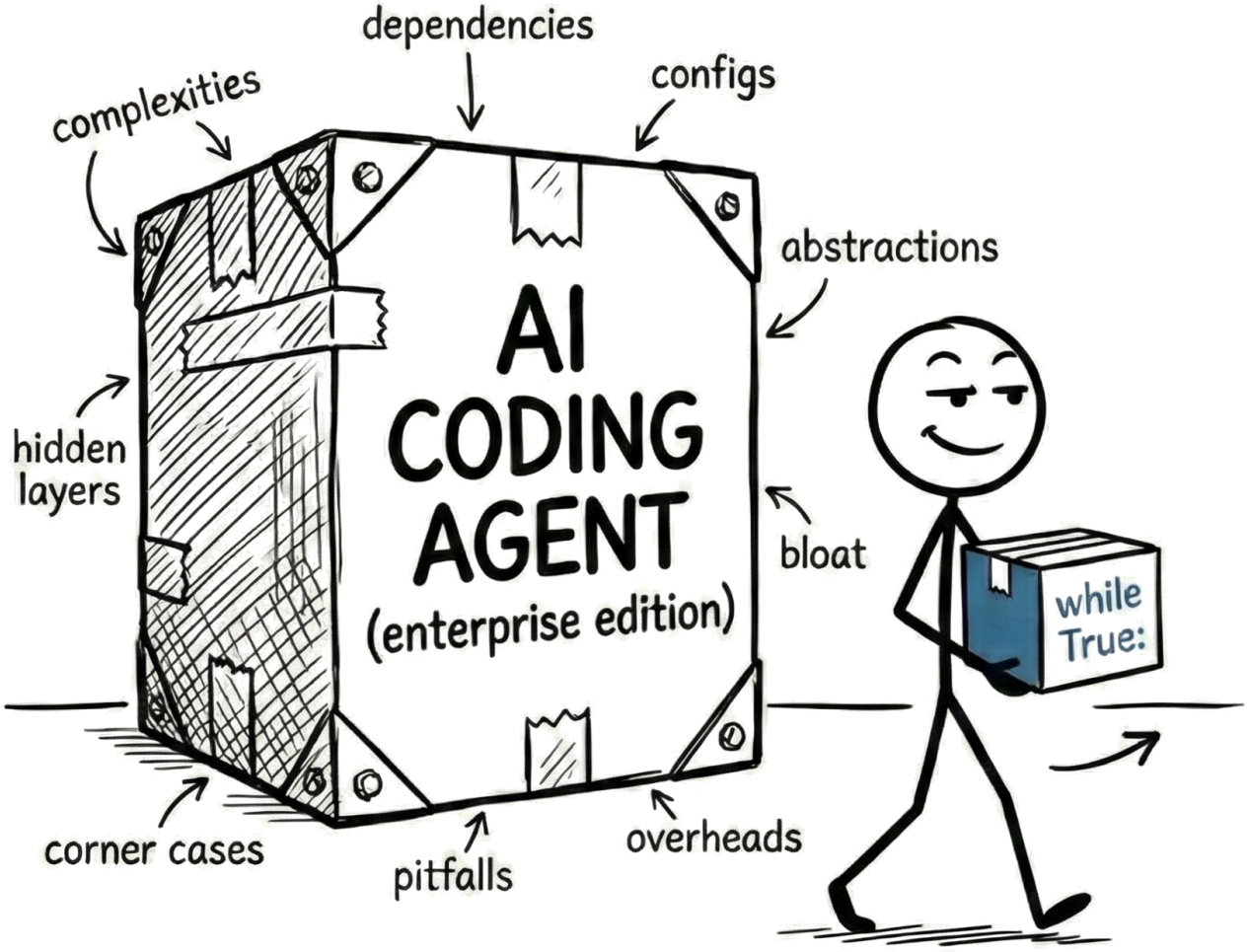
Build Your Own
CODING AGENT

The Zero-Magic Guide to AI Agents in Pure Python



J. Owen

Build Your Own Coding Agent

The Zero-Magic Guide to AI Agents in Pure Python

J. Owen

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

This version was published on 2026-11-05



© 2026 J. Owen

Tweet This Book!

Please help J. Owen 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.

Table of Contents

Preface **1**
Who This Book Is For 1
What You Will Build 1
Testing Approach 2
Code Examples 2
Conventions Used in This Book 3

Part I: The Brain **4**

Chapter 1: The Zero Magic Manifesto **5**
What is an Agent, Really? 5
What We Are Building 6
Project Setup 7
The AgentStop Exception 9
The Agent Class 9
Defining Success with Tests 10
The Main Loop 12
Run It 13
Wrapping Up 13

Chapter 2: The Raw Request **14**
Get an API Key 14
The Vault (.env) 14
The Anatomy of a Request 15
The Code 16
Run It 18
Troubleshooting 18
Cleaning Up 19
Wrapping Up 20

Chapter 3: The Infinite Loop **21**

The Illusion of Memory . . . 21
The Testing Problem . . . 22
Response Types . . . 23
The FakeBrain Pattern . . . 24
Defining Success . . . 25
The Claude Class . . . 27
The Agent Class (Updated) . . . 30
The Main Loop (Updated) . . . 31
Verify the Tests Pass . . . 32
Test the Memory . . . 32
The Context Window Problem . . . 33
Wrapping Up . . . 34

Chapter 4: The Universal Adapter . . . 35
The Adapter Pattern . . . 35
HTTP Resilience . . . 35
The Brain Interface . . . 35
The FakeBrain (Updated) . . . 35
The Claude Brain (Refactored) . . . 35
The DeepSeek Brain . . . 36
The BRAINS Registry . . . 36
The Agent Class (Updated) . . . 36
Tests for Multi-Brain Support . . . 36
The Main Loop (Updated) . . . 36
Setting Up DeepSeek . . . 36
Try It . . . 36
“We Just Moved Code Around” . . . 37
Wrapping Up . . . 37

Part II: The Hands . . . 38

Chapter 5: The Tool Protocol . . . 39
How Tools Actually Work . . . 39
Defining the Tool Interface . . . 39
The ReadFile Tool . . . 39
The WriteFile Tool . . . 39
Tool Helpers . . . 39
Updating the Thought Class . . . 40
Updating the Claude Class . . . 40

The Agent Class with Tools 40
The Main Loop 40
Test It 40
Wrapping Up 40

Chapter 6: The Scratchpad (Memory) **41**
The "Zero Magic" Memory 41
The Memory Class 41
The ToolContext Class 41
The SaveMemory Tool 41
Updating the Claude Class 41
Crafting the System Prompt 42
Updating the Agent Class 42
The Main Loop (Updated) 42
Testing Persistence 42
Wrapping Up 42

Chapter 7: The Safety Harness (Plan Mode) **43**
The Concept 43
Tests First 43
The WritePlan Tool 43
One List, Two Views 43
Telling the Brain What Mode It's In 43
The Agent Class (Updated) 44
The Main Loop (Updated) 44
Testing the Harness 44
The Psychology of the "Plan" 44
Wrapping Up 44

Chapter 8: The Context Pipeline (Map & Search) **45**
The ListFiles Tool 45
The SearchCodebase Tool 45
Update the Tools List 45
The "Zoom In" Test 45
Wait, is this RAG? 45
Wrapping Up 46

Chapter 9: The Reality Check (Running Code) **47**
The Feedback Loop 47
Tests First 47

The RunCommand Tool 47
The Interactive Trap 47
The Self-Healing Demo 47
The TDD Workflow 48
The Surgical Edit 48
The Closed Loop 48
Hardening the Loop 48
Context Compaction 48
Security Considerations 48
Wrapping Up 48

Part III: The Frontier **49**

Chapter 10: Going Dark (Local Models) **50**
The Trade-off 50
Installing Ollama 50
The Ollama Brain Class 50
Running with Ollama 50
The “Infinite Loop” Experiment 50
The Practical Differences 51
The Hybrid Workflow 51
Model Selection 51
Troubleshooting Ollama 51
Wrapping Up 51

Chapter 11: The Extension (Web Search) **52**
Step 1: The Meta-Prompt 52
Step 2: The Surgery 52
Step 3: The Reference Implementation 52
Step 4: The Tests 52
Self-Modification 52
Wrapping Up 53

Chapter 12: The Capstone (Building a Game) **54**
Step 1: Preparation 54
Step 2: The Architect (Plan Mode) 55
Step 3: The Builder (Act Mode) 56
Step 4: The Reality Check 57
Step 5: The Pivot (Feature Creep) 58
Check the Result 58

What Goes Wrong 59
Wrapping Up 60
Epilogue 60

Appendix A: Streaming Responses 61
How Streaming Works 61
The Implementation 61
What Changed from Chapter 11 61
The Trade-Off 61
Running the Code 61

Acknowledgments 62

Preface

Who This Book Is For

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

You have tried the frameworks—and watched your LangChain app hallucinate its way into deleting a production database. And you think: "*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
- Separate planning from execution with a safety mode
- Search the web for documentation and answers

You will build it from scratch using `requests`, `python-dotenv`, and `pytest`; shell access comes from Python's standard-library `subprocess` module. No LangChain. No vector databases. No "orchestration frameworks." Just Python you can debug with `print()`.

Chapter 11 adds `ddgs` for web search. The Snake capstone also uses `pygame-ce`, which supplies `import pygame`; it is separate from the agent's dependencies. Use Python 3.15 for this edition, with 3.12 as the tested minimum. The setup pins the package versions used to verify these examples.

Testing Approach

This book uses a "test-alongside" approach. For most features:

1. We introduce the concept—why we need this
2. We show the test first, so you know what success looks like
3. We implement the code to make the test pass
4. We verify with pytest

This teaches *Test-Driven Development* thinking without the full red-green-refactor ceremony in print. Live API calls make unit tests slow, expensive, and unpredictable. Keep those calls for separate integration tests and model evaluations.

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/`, `ch03/`, `ch04/`, etc.) plus an `appendix/` folder. Most folders contain:

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

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

Two exceptions: `ch02/` contains only a standalone `test_api.py` script (used once and discarded), and `ch12/`'s agent snapshot is functionally the same as `ch11/`. You generate the capstone game while working through Chapter 12.

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

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.

Sample sessions are illustrative. Thinking and progress messages may be shortened, and bracketed tool labels annotate actions rather than reproduce the exact console format. Tool results shown in a session explain what the model receives; the non-streaming agent does not print every tool's returned content directly.

Time to write some code.

Part I: The Brain

Part I builds the skeleton of an agent: a `while` loop that talks to Claude via raw HTTP. By Chapter 4, you'll support multiple LLM providers through the Adapter Pattern—and understand that an "AI agent" is just a loop, a brain, and a list of messages.

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 take the "Zero Magic" approach: build a working coding-agent foundation called **Nanocode** in pure Python. No LangChain, no AutoGPT, no 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.

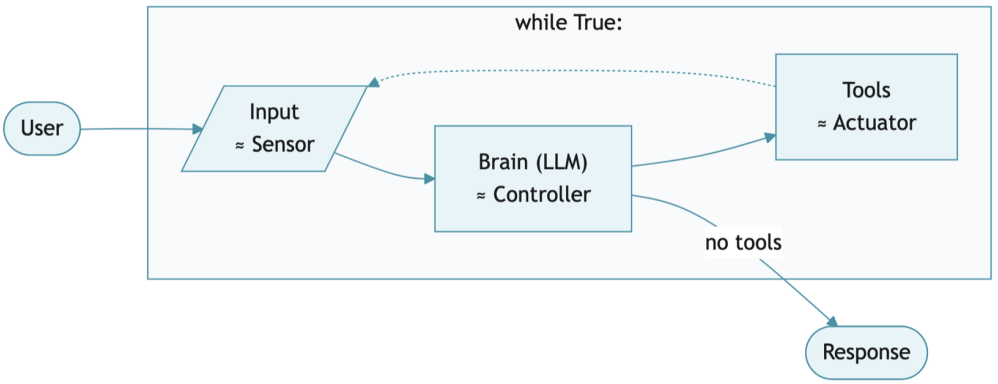


Figure 1. The Agent Loop: User input flows through a while loop where Input (Sensor) feeds the Brain/LLM (Controller), which triggers Tools (Actuator), cycling back to Input until the Brain outputs a Response.

More specifically, an agent has four parts. The *Brain* is the LLM—a stateless function where you send text and it returns text. It calls *Tools*—functions like Read File and Run Command—to interact with the outside world. All of this sits inside a *Loop* (a `while True`) that keeps cycling until the task is done, with *Memory*—just a Python list—accumulating the conversation history along the way. (The list 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 wired it up to Claude Sonnet 5.5 (or DeepSeek, or a local model via Ollama). You'll give it hands—tools to read files, write files, and run shell commands—and eyes to search your codebase. And you'll separate planning from execution so the agent can inspect a problem before you enable file changes and shell commands. Execution mode still grants real filesystem and shell access; it is not a sandbox.

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

Use Python 3.15 for this edition; Python 3.12 is the tested minimum.[1] Check `python3 --version` before creating the environment. On Windows, check `python --version` instead. A virtual environment keeps project packages separate from 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:

[1] https://www.python.org/downloads/

- `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==2.34.2
python-dotenv==1.2.4
pytest==9.2.0
```

These pins are the versions tested for this edition, not a claim that other versions fail. Chapter 11 adds the search package. Install into the active environment:

```
python -m 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 exit mechanism. An exception beats `break` statements scattered across the code.

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:

```
# --- Exceptions ---

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

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

The Agent Class

Now the core abstraction: the `Agent` class. It holds state and logic in one place, which makes it easy to test.

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:

```
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:** The `/q` command raises `AgentStop` instead of returning a special value—the caller decides how to handle shutdown.
- **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, we need tests.

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 break out of the loop.



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

That's the chassis: an `Agent` class, a `handle_input()` method, a `while True` loop. It doesn't do anything useful yet—but everything we build from here plugs into this skeleton. The tests make sure we don't break what already works as we go.

Chapter 2: The Raw Request

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

SDKs hide the truth. They add layers of abstraction that make "Hello World" easy but make debugging "Error 400" a nightmare. When you learn the SDK, you learn the SDK. When you learn the raw HTTP call, you learn the protocol underneath every SDK.

We are going to send a message to Claude using nothing but the `requests` library. Claude is Anthropic's model family; we use Sonnet 5.5 here.

Get an API Key

To talk to Claude, you need an API Key. This is a long string of characters that acts like a password tied to your billing account.

1. Go to the Anthropic Console.[1]
2. Sign up and buy API credits through the Console's billing page.
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.

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`.

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: authentication in the headers (your API key), configuration in the body (which model, how many tokens), and the message itself.

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).

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:

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

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

# Fail early with a clear message instead of an API authentication error
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-5-5",
    "max_tokens": 4096,
    "thinking": {"type": "between_tools"},
```

```
30      "output_config": {"effort": "medium"},
31      "messages": [
32          {"role": "user", "content": "Hello, are you ready to code?"}
33      ]
34  }
35
36  # 5. Fire! (No safety net)
37  print("📡 Sending request to Claude...")
38  response = requests.post(url, headers=headers, json=payload, timeout=120)
39
40  # 6. Inspect the raw result
41  print(f"Status: {response.status_code}")
42
43  if response.status_code == 200:
44      # Success: Print the beautiful JSON
45      print("Response:")
46      print(json.dumps(response.json(), indent=2))
47  else:
48      # Failure: Print the ugly raw text so we can debug
49      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:** Fail early with a clear configuration error if the key is missing. For this `requests.post()` call, Requests removes the `None` header value, so the API would receive a request without its API-key header and reject authentication.
- **Line 21:** The `anthropic-version` header selects the API contract. Keep `2023-06-01`; it is not the model's release date.
- **Line 27:** `claude-sonnet-5-5` specifies which model we want.
- **Line 28:** `max_tokens` caps this response at 4,096 output tokens. It does not cap spending across repeated calls.
- **Lines 29-30:** `between_tools` with medium effort keeps this no-tools example focused on text. Claude uses thinking between tool calls in this mode; we introduce adaptive thinking with visible summaries when the agent gains a brain.[2]
- **Line 38:** We fire the request with a 2-minute timeout. No try/except—if the network is down, let Python crash. You need to see where it fails.

[2] https://platform.claude.com/docs/en/models/sonnet-5-5/migration-guide

- **Lines 43-49:** Check the status code. 200 means success (pretty-print the JSON). Anything else means we print the raw error text for debugging.

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	Missing or invalid API key	Check `.env` is loading with `bool(os.getenv("ANTHROPIC_API_KEY"))`. This checks presence without printing the secret; a present key can still be invalid.
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're done with it.



Aside: 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: As checked on November 4, 2026, Sonnet 5.5 costs $2 per million ordinary input tokens and $10 per million output tokens.[3] A request with 20,000 uncached input tokens and 2,000 output tokens costs $0.06 at those rates. Cache reads and writes have different prices; reasoning consumes output tokens. Check current rates before running a long task.



Aside: Check the Anthropic Console's Usage tab to monitor spending. The response's `usage` object reports token counts. A session's bill depends on repeated history, cached input, output, and thinking; the number of exchanges alone cannot predict it.

Wrapping Up

That's a raw API call: headers, JSON payload, response parsing. No abstraction between you and the wire. When something breaks (and it will), you'll know exactly which layer failed because there's only one layer.

One problem: Claude has total amnesia. Every request is a blank slate. We're going to fake memory by replaying the entire conversation history on every turn.

[3]https://platform.claude.com/docs/en/about-claude/pricing

Chapter 3: The Infinite Loop

We have a problem.

Imagine running 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 by creating 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.

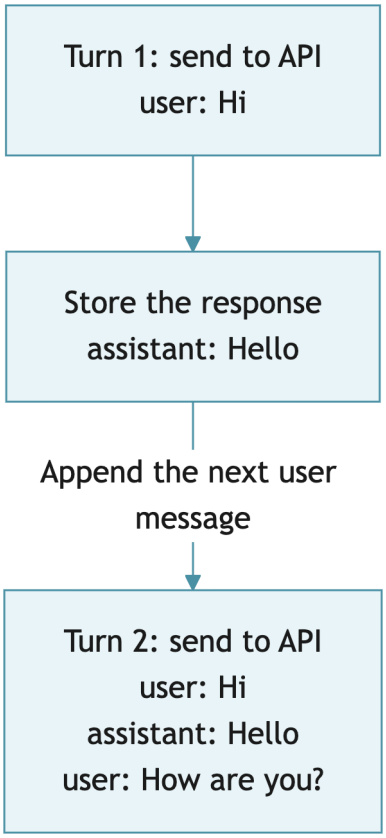


Figure 2. Context accumulation: Turn 1 sends just "User: Hi" to the API. Turn 2 sends the full history—"User: Hi", "Assistant: Hello", "User: How are you?"—to the API.

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

Let's implement this *context loop* manually. But first, we need to make our code testable.

The Testing Problem

Calling a live LLM on every unit-test run makes the suite slow and unpredictable. Keep those calls for separate integration tests and model evaluations.

API calls are slow (2–10 seconds each), expensive (real money per call), and nondeterministic (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.

We'll establish this pattern 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 plain data objects to represent these possibilities. (We skip `@dataclass` deliberately—these classes are simple enough that the decorator would save a few lines while hiding what `__init__` actually does.)

The Code:

```
17  class ToolCall:
18      """A tool invocation request from the brain."""
19
20      def __init__(self, id, name, args):
21          self.id = id
22          self.name = name
23          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` yet—the brain can't call tools—but we define it now because it's part of the `Thought` response type. When we add tools, Claude will return these when it wants to read a file or execute a command.

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

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

A `Thought` is what the brain returns after thinking. It might have text, tool calls, both, or neither. The `thinking` field captures the model's reasoning summary—we'll see where that comes from when we build the Claude class below. 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:

```
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. Update that file's imports to include the response types:

```
import pytest
from nanocode import Agent, AgentStop, Thought, ToolCall
```

Notice that `FakeBrain` has the same interface as our real brain will—a `think()` method that takes a conversation and returns a `Thought`.

Keep the Chapter 1 tests. The updated `Agent` will require a brain, so change their constructors to `Agent(brain=FakeBrain())`. For the string-response test, give the fake a response containing `hello` so both existing assertions still apply:

```
def test_handle_input_returns_string():
    agent = Agent(brain=FakeBrain(responses=[Thought(text="hello")]))
    result = agent.handle_input("hello")
    assert isinstance(result, str)
    assert "hello" in result
```



Aside: This pattern—replacing a real dependency with a predictable fake for testing—is called *test doubling*. 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

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

```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
```

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

```
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

Now the real brain. Add these imports and the environment loading call at the top of `nanocode.py`:

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

load_dotenv()
```

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`. We also enable *adaptive thinking*: Claude decides when to reason and how much work a request needs. Think of it as a notepad used before speaking; the visible summary is a report of that work, not a complete transcript. Reasoning consumes output tokens and can help with tasks such as choosing tools.

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-5-5"
        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": 16000,
            "thinking": {"type": "adaptive", "display": "summarized"},
            "output_config": {"effort": "medium"},
            "messages": conversation
        }

        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 41-43:** Load the API key and fail fast if it's missing.
- **Lines 44-45:** Store config. We'll make the model configurable later.
- **Line 47:** The `think()` method is the brain's interface—same as `FakeBrain`.
- **Lines 55-57:** `adaptive` lets Claude decide when to think. `display: summarized` requests a visible reasoning summary; Sonnet 5.5 otherwise omits it by default. Medium effort controls reasoning depth, not a fixed token allowance. The 16,000-token output cap covers reasoning and response text together; thinking is billed as output.[2]
- **Line 58:** The payload includes `"messages": conversation`—the full history, not just the current message. This is the context loop.
- **Line 63:** Parse Claude's complex response format into our simple `Thought`.

Now the response parser:

```
65      def _parse_response(self, content):
66          """Convert Claude's response format to Thought."""
67          text_parts = []
68          tool_calls = []
69          thinking = None
70
71          for block in content:
72              if block["type"] == "thinking":
73                  thinking = block["thinking"]
74              elif block["type"] == "text":
75                  text_parts.append(block["text"])
76              elif block["type"] == "tool_use":
77                  tool_calls.append(ToolCall(
78                      id=block["id"],
79                      name=block["name"],
80                      args=block["input"]
81                  ))
82
83          return Thought(
84              text="\n".join(text_parts) if text_parts else None,
85              tool_calls=tool_calls,
86              thinking=thinking
87          )
```

Claude's API returns a list of "content blocks." Each block has a `type`—`"thinking"`, `"text"`, or `"tool_use"`. A thinking block may contain a reasoning summary—we store it on the `Thought` for the caller to display. It can also be

[2] https://platform.claude.com/docs/en/models/sonnet-5-5/migration-guide

empty or absent, so the display must not assume visible reasoning exists. Text blocks become the response, and `tool_use` blocks become `ToolCall` objects. The parser doesn't print anything; it just converts the raw JSON into a clean `Thought`.

The Agent Class (Updated)

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

The Code:

```
92  class Agent:
93      """A coding agent with conversation memory."""
94
95      def __init__(self, brain):
96          self.brain = brain
97          self.conversation = []
98
99      def handle_input(self, user_input):
100         """Handle user input. Returns output string, raises AgentStop to
            ↪ quit."""
101         if user_input.strip() == "/q":
102             raise AgentStop()
103
104         if not user_input.strip():
105             return ""
106
107         self.conversation.append({"role": "user", "content": user_input})
108
109         try:
110             thought = self.brain.think(self.conversation)
111             if thought.thinking:
112                 lines = thought.thinking.strip().split("\n")[:5]
113                 for i, line in enumerate(lines):
114                     prefix = "  💭 " if i == 0 else "     "
115                     print(f"\033[2m{prefix}{line}\033[0m")
116             text = thought.text or ""
117             self.conversation.append({"role": "assistant", "content": text})
118             return text
119         except Exception as e:
120             self.conversation.pop()  # Remove failed user message
121             return f"Error: {e}"
```

The Walkthrough:

- **Lines 95–97:** Accept a brain via dependency injection. Initialize an empty conversation list.
- **Line 107:** Append the user's message to history *before* calling the brain.
- **Lines 110–118:** Call the brain, display up to five lines of thinking in dimmed text (`\033[2m` is the ANSI escape code for dim, `\033[0m` resets it), extract the response, and append it to history.
- **Lines 119–121:** If the API call fails, remove the user message we just added. This keeps the conversation in a valid state.

Pay attention to line 107: 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:

```
126 def main():
127     brain = Claude()
128     agent = Agent(brain)
129     print("⚡ Nanocode v0.2 (Conversation Memory)")
130     print("Type '/q' to quit.\n")
131 
132     while True:
133         try:
134             user_input = input("❯ ")
135             output = agent.handle_input(user_input)
136             if output:
137                 print(f"\n{output}\n")
138 
139         except (AgentStop, KeyboardInterrupt):
140             print("\nExiting...")
141             break
142 
```

```

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.
  💭 The user is telling me about their project. They want to build
     a Python agent. I should respond helpfully and ask what kind
     of agent they're building.

That sounds exciting! What kind of agent are you building?

❯ What language am I using?
  💭 The user previously said they are building a Python agent.
     The answer is Python.

You are using Python.
```

The conversation list is doing its job.

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*: the desk space available for the current request and the output it generates. The capacity depends on the model. If the input alone exceeds Claude's limit, the API returns `400 Bad Request`. Generation can also reach the window boundary; newer Claude models then return `stop_reason: "model_context_window_exceeded"` rather than a 400.[3]

Our error handling on line 119 reports failed requests and removes the latest user message. A shorter next message may fit if the failed message caused the

[3]https://platform.claude.com/docs/en/build-with-claude/context-windows

overflow. If the retained history alone is too long, every new request will still fail until we reduce that history.

For now, restarting the agent clears the history and gets you back on track. We'll add proper *context compaction*—tracking token usage from the API response and automatically summarizing old messages before they overflow—when we build the feedback loop in Chapter 9. That's where conversations actually blow up, and where the fix will earn its keep.

> **Dev Tip:** The request blocks until the full response arrives. The thinking display on lines 111-115 appears afterward; it shows the reasoning summary but provides no feedback during the wait. An activity message printed before the call would help users distinguish a pending request from a frozen program. Streaming can show output as it arrives; the appendix adds that later.

Wrapping Up

Claude remembers now—or rather, we've tricked it into thinking it does. The conversation list grows with every turn, and `FakeBrain` lets us test the whole thing without spending a cent.

Both patterns will carry through the rest of the book. Every brain we build (Claude, DeepSeek, Ollama) will implement the same `think()` interface, and `FakeBrain` will test them all.

One loose end: our code is hardwired to Anthropic's API. If we want to add DeepSeek or a local model, we'd have to duplicate a lot of code.

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.

“We Just Moved Code Around”

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.

Part II: The Hands

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

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.

How Tools Actually Work

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.

Defining the Tool 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 ReadFile 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 WriteFile 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.

Tool Helpers

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 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

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 Agent 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 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 Persistence

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 7: The Safety Harness (Plan Mode)

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 Concept

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 First

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 WritePlan 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.

One List, Two Views

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.

Telling the Brain What Mode It's In

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.

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.

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)

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 Feedback 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.

Tests First

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 RunCommand 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 Interactive Trap

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 Self-Healing Demo

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 TDD Workflow

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 Surgical Edit

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 Closed 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.

Hardening the 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.

Context Compaction

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.

Security Considerations

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.

Part III: The Frontier

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 10: Going Dark (Local Models)

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 Trade-off

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.

Installing Ollama

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 Ollama Brain 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.

Running with Ollama

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 “Infinite Loop” Experiment

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 Practical Differences

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 Hybrid Workflow

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.

Model Selection

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.

Troubleshooting Ollama

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 11: The Extension (Web 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.

Step 1: The Meta-Prompt

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.

Step 2: The Surgery

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.

Step 3: The Reference Implementation

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.

Step 4: The Tests

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.

Self-Modification

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 12: The Capstone (Building a Game)

Can Nanocode actually build something?

The "Zero Code Challenge": 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

From your project root (the directory containing `nanocode.py` and `.env`), create a working directory for the game:



Aside: We work in a subdirectory to keep the game files together and start with a fresh memory. The copy of `nanocode.py` is still visible to the agent.

```
mkdir -p snake_game
cp nanocode.py snake_game/
cp .env snake_game/
cd snake_game
```

If you're starting from the root of the reader code repository, copy from `ch11` instead (create your local `.env` first):

```
mkdir -p snake_game
cp ch11/nanocode.py snake_game/
cp .env snake_game/
cd snake_game
```

Install pygame-ce, the maintained Pygame Community Edition used for this example. It supplies the same `import pygame` namespace.[1] Use a fresh environment so the original `pygame` distribution and `pygame-ce` do not overwrite each other:

```
python -m venv venv
source venv/bin/activate
python -m pip install requests==2.34.2 python-dotenv==1.2.4 pytest==9.2.0 \
  ddgs==9.16.1 pygame-ce==2.5.8
```

Keep `import pygame` in `snake.py`. `pygame-ce` is a capstone dependency, not part of Nanocode's core requirements.



Aside: The shell commands above are for macOS/Linux. On Windows, use `mkdir snake_game`, `copy nanocode.py snake_game\`, and `copy .env snake_game\`, then activate the environment with `venv\Scripts\activate`. pygame-ce 2.5.8 provides Python 3.15 wheels for common platforms.[2] If pip attempts a source build, check your Python version and platform against the available wheels before installing SDL development packages.

Step 2: The Architect (Plan Mode)

Set a longer command timeout before starting the agent. Pygame's main loop runs until you close the window, so the default 30 seconds leaves little time to play. The value is in seconds and is inherited by the Nanocode process when it starts. We begin in plan mode because we want a blueprint before we lay bricks.

[1] https://pyga.me/docs/
[2] https://pypi.org/project/pygame-ce/2.5.8/#files

```
export NANOCODE_TIMEOUT=300
python nanocode.py
```

The Prompt:

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

The agent will use `write_plan` 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 the plan looks reasonable, move on to Step 3.

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

The agent already has the five-minute timeout set before startup. Close the game window when you finish testing so Nanocode can continue. Exporting a different value in another terminal would not change this running agent's environment.

The Prompt:

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

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



Warning: If the command times out, the game process may still be running even though the tool returned an error. The model can then try another command before you regain the prompt. If it starts repeating launches, interrupt Nanocode, close surviving game windows or stop their processes, and inspect the files before restarting. The timeout does not provide process-tree cleanup.

> **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 it again.

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

Check the Result

An open window is only the first check. Play through these cases and report each failure to the agent with what you did, what happened, and what you expected:

- Arrow keys change direction; reversing directly into the snake's own body is handled consistently.
- Eating food grows the snake and increments the score once. New food appears on an empty cell.
- Wall and self collisions end the round and display Game Over.
- Restart resets the score, snake, food, and speed; repeated restarts still work.
- After the feature change, food changes the snake's color and each five-point boundary increases speed once.
- Closing the window exits cleanly and returns control to Nanocode without a traceback.

The agent cannot see the window. A zero exit code and a passing unit test cannot establish these visual and interactive behaviors for you. Keep a short record of the checks you performed and inspect `git diff` before accepting the result.

What Goes Wrong

Your results will differ from mine—LLMs are non-deterministic. But here's what typically happens, and what to watch for.

Common failures on the first run:

- `ModuleNotFoundError: No module named 'pygame'` — the agent forgot you need to install it, or it ran the script in a different environment. Run `python -m pip install pygame-ce==2.5.8` in the same environment used to launch the game. If both Pygame distributions were installed together, recreate the clean environment above.
- `AttributeError` on a method the agent defined but misspelled in the call site. The agent fixes these quickly once it sees the traceback.
- Off-by-one errors in collision detection. The snake walks through walls or dies one pixel too early. These take 2-3 iterations of edit-run-fix.

The typical session arc:

In my tests, the agent usually gets a working (but ugly) game in 2-4 iterations. The first write produces something that crashes. The second or third fix gets it running. The feature creep step (color changes, speed ramps) adds another 3-5 iterations as the agent reads its own code, makes surgical edits, and verifies each change.

Long sessions may trigger "(Compacting conversation...)" when the measured input-token count exceeds 75% of the brain's configured context limit. The number of rounds depends on how much text and tool output each round contains; there is no fixed round count. After compaction, the agent loses some detail about early rounds but keeps working. For Ollama, the adapter reads the active served window before each request, as discussed in Chapter 10.

Where the agent struggles:

Pygame's coordinate system and event loop are tricky. The agent sometimes writes code that renders correctly but doesn't handle keyboard input properly, or that draws the snake in the wrong order so the head appears behind the body. These are the kind of bugs a human spots instantly but the agent can't see—it has no visual feedback, only stdout and stderr. If the game runs without errors but looks wrong, you'll need to describe the visual bug: "The snake renders backwards—the head should be at the front."

The point isn't perfection on the first try. It's that the agent *converges*—write, run, read the error, fix, repeat—using every tool we built across eleven chapters.

Before you ship the final `snake.py`, read it. The agent writes code that *runs*, but a human should still audit for things the test loop can't catch: hardcoded magic numbers, missing edge cases (what if the window is resized?), and whether the code is structured in a way you'd want to maintain. The agent is a fast drafter, not a final reviewer.

Wrapping Up

Plan, implement, crash, debug, fix, run again—that's every chapter earning its keep.

So where does it go from here?

Epilogue

The whole thing is about 800 lines of Python. No framework. `nanocode.py` is yours. Do whatever you want with it:

- Git integration that auto-commits after green tests
- Screenshot-based debugging for frontend work (Claude can read images)
- Voice input via Whisper so you can talk instead of type
- MCP for connecting to external services your team already uses
- Sub-agents that fork conversations and work on subtasks in parallel

Production agents like Claude Code, Cursor, and Copilot do more than this: streaming responses (see Appendix A), parallel tool execution, tree-sitter parsing, sandboxed execution environments, multi-file context windows spanning thousands of files. The gap between 800 lines and a large production system is real. But the architecture is the same: a brain, a loop, tools, memory, and a safety harness. Now you know what's behind the curtain.

The models will keep getting better. The harness—the loop, the tools, the safety checks—that part is engineering. And that part isn't going anywhere.

Appendix A: Streaming Responses

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.

How Streaming Works

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 Implementation

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.

What Changed from Chapter 11

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 Trade-Off

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.

Running the Code

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.

Acknowledgments

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.