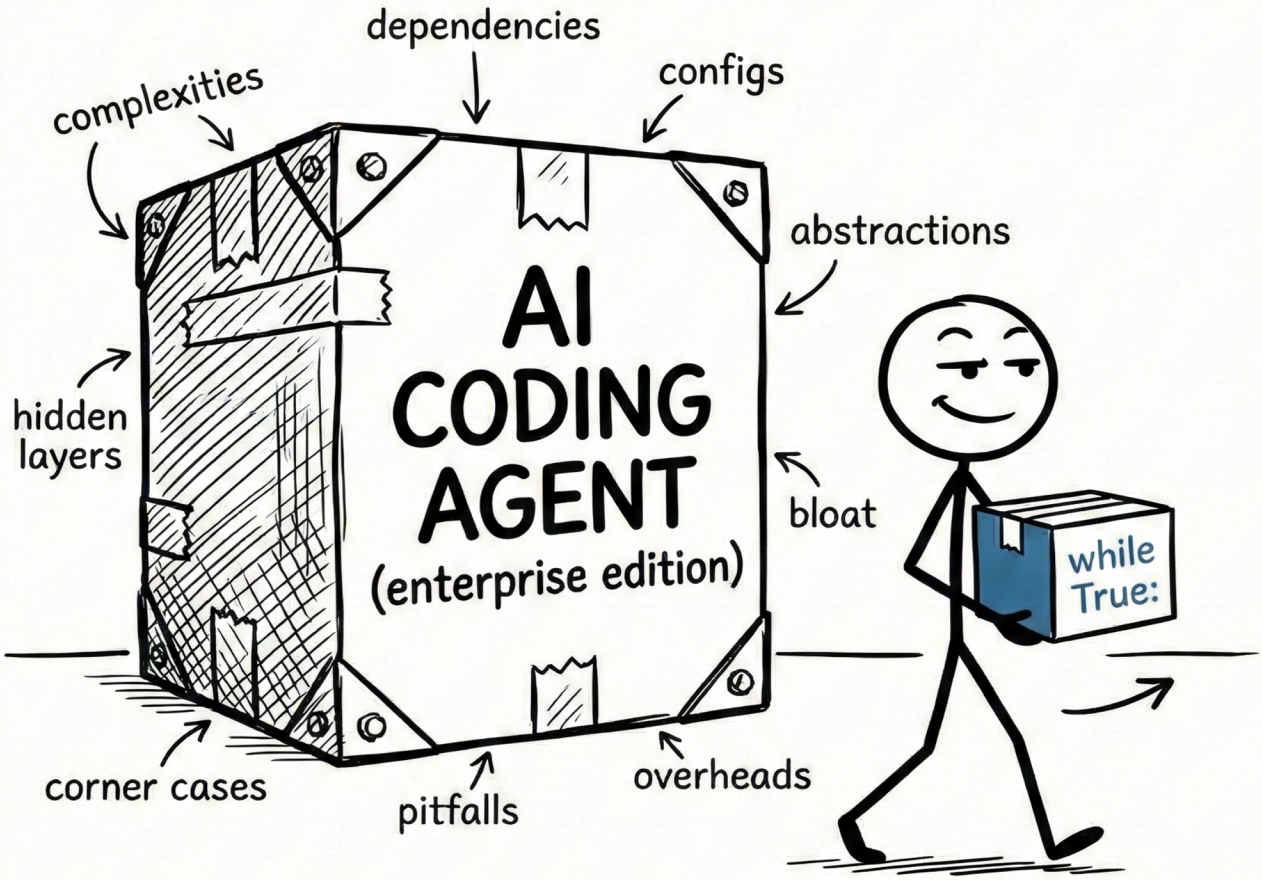
Build Your Own
CODING AGENT

The Zero-Magic Guide to AI Agents in Pure Python



Owen Ou

Edición en Español

Construye tu Propio Agente de Programación (Spanish Edition)

Guía Sin-Magia para Agentes de IA en Python Puro

Owen Ou y TranslateAI

Este libro está a la venta en https://leanpub.com/build-your-own-coding-agent-es

Esta versión se publicó en 2026-02-18

© 2026 Owen Ou y TranslateAI

¡Tuitea sobre el libro!

Por favor ayuda a Owen Ou y TranslateAI hablando sobre el libro en Twitter!

El tuit sugerido para este libro es:

Acabo de construir un agente de código desde cero usando solo llamadas HTTP y un shell. Resulta que la ”magia de la IA” no es más que un bucle while y una llamada a la API.

El hashtag sugerido para este libro es #buildyourowncodingagent.

Descubre lo que otra gente dice sobre el libro haciendo clic en este enlace para buscar el hashtag en Twitter:

#buildyourowncodingagent

A mi esposa, mi hijo, mis padres y mis abuelos—ustedes me enseñaron que no existe la magia, solo el esfuerzo. Todo lo que construyo, lo construyo gracias a ustedes.

Índice general

Prefacio **1**
Para Quién es Este Libro 1
Lo Que Vas a Construir 1
Enfoque de Pruebas 1
Ejemplos de Código 2
Convenciones Utilizadas en Este Libro 2

Parte I: El Cerebro **4**

Capítulo 1: El Manifiesto Cero Magia **5**
¿Qué es realmente un Agente? 5
Qué Vamos a Construir 6
Configuración del Proyecto 6
La Excepción AgentStop 8
La Clase Agent 9
Definiendo el éxito con pruebas 10
El Bucle Principal 11
Ejecútalo 12
Conclusión 12

Capítulo 2: La Solicitud sin Procesar **13**
Obtener una Clave API 13
La Bóveda (.env) 13
La Anatomía de una Solicitud 14
El Código 15
Ejecútalo 16
Resolución de problemas 17
Limpieza 17
Conclusión 18

Capítulo 3: El Bucle Infinito **19**

La Ilusión de la Memoria . . . 19
El Problema de las Pruebas . . . 19
Tipos de Respuesta . . . 20
El Patrón del Cerebro Falso . . . 21
Definiendo el Éxito . . . 22
La Clase Claude . . . 23
La Clase Agent (Actualizada) . . . 25
El bucle principal (Actualizado) . . . 27
Verificar que las Pruebas Pasen . . . 27
Probar la Memoria . . . 28
El Problema de la Ventana de Contexto . . . 28
Conclusión . . . 29

Capítulo 4: El Adaptador Universal . . . 30
El Patrón Adaptador . . . 30
Resiliencia HTTP . . . 30
La Interfaz Brain . . . 30
El FakeBrain (Actualizado) . . . 30
El Cerebro Claude (Refactorizado) . . . 30
El Cerebro DeepSeek . . . 30
El Registro de BRAINS . . . 31
La Clase Agent (Actualizada) . . . 31
Pruebas para Soporte Multi-Cerebro . . . 31
El Bucle Principal (Actualizado) . . . 31
Configuración de DeepSeek . . . 31
Pruébalo . . . 31
“Solo Movimos Código” . . . 31
Conclusión . . . 32

Parte II: Las Manos . . . 33

Capítulo 5: El Protocolo de Herramientas . . . 34
Cómo Funcionan Realmente las Herramientas . . . 34
Definiendo la Interfaz de Herramientas . . . 34
La Herramienta ReadFile . . . 34
La Herramienta WriteFile . . . 34
Funciones Auxiliares de Herramientas . . . 34
Actualizando la Clase Thought . . . 34
Actualizando la Clase Claude . . . 35

La Clase Agent con Herramientas 35
El Bucle Principal 35
Pruébalo 35
Conclusión 35

Capítulo 6: El Scratchpad (Memoria) **36**
La Memoria "Cero Magia" 36
La Clase Memory 36
La Clase ToolContext 36
La Herramienta SaveMemory 36
Actualizando la Clase Claude 36
Elaboración del Prompt del Sistema 36
Actualizando la Clase Agent 37
El Bucle Principal (Actualizado) 37
Prueba de Persistencia 37
Conclusiones 37

Capítulo 7: El Arnés de Seguridad (Modo Planificación) **38**
El Concepto 38
Primero las Pruebas 38
Extendiendo ToolContext 38
La Herramienta WriteFile Protegida 38
La Clase Agent (Actualizada) 38
El Bucle Principal (Actualizado) 38
Probando el Arnés de Pruebas 39
La Psicología del "Plan" 39
Conclusión 39

Capítulo 8: El Pipeline de Contexto (Mapeo y Búsqueda) **40**
La Herramienta ListFiles 40
La Herramienta SearchCodebase 40
Actualizar la Lista de herramientas 40
La Prueba "Zoom In" 40
Espera, ¿esto es RAG? 40
Conclusión 40

Capítulo 9: La Prueba de Realidad (Ejecutar Código) **42**
El Ciclo de Retroalimentación 42
Primero las Pruebas 42
La Herramienta RunCommand 42

La Trampa Interactiva 42
La Demostración de Auto-Reparación 42
El flujo de trabajo de TDD 42
La Edición Quirúrgica 43
El Bucle Cerrado 43
Compactación del Contexto 43
Consideraciones de Seguridad 43
Conclusión 43

Parte III: La Frontera 44

Capítulo 10: Desconectándose (Modelos locales) 45
El compromiso 45
Instalación de Ollama 45
La Clase Cerebro de Ollama 45
Ejecutando con Ollama 45
El Experimento del "Bucle Infinito" 45
Las Diferencias Prácticas 45
El Flujo de Trabajo Híbrido 46
Selección de Modelos 46
Resolución de problemas de Ollama 46
Conclusión 46

Capítulo 11: La Extensión (Búsqueda Web) 47
Paso 1: El Meta-Prompt 47
Paso 2: La Cirugía 47
Paso 3: La Implementación de Referencia 47
Paso 4: Las Pruebas 47
Automodificación 47
Conclusión 47

Capítulo 12: El Proyecto Final (Construyendo un Juego) 49
Paso 1: Preparación 49
Paso 2: El Arquitecto (Modo de Planificación) 50
Paso 3: El Constructor (Modo de Acción) 50
Paso 4: La Verificación de la Realidad 51
Paso 5: El Giro (Expansión de Características) 52
Lo Que Sale Mal 52
Conclusión 53
Epílogo 53

Agradecimientos **55**

Prefacio

Para Quién es Este Libro

Eres un ingeniero de software escéptico del bombo publicitario de la IA.

Has visto las demostraciones. Has probado los frameworks. Has visto cómo tu aplicación de LangChain alucina hasta borrar una base de datos en producción. Y piensas: *"Tiene que haber una mejor manera."*

La hay. Este libro es para desarrolladores que quieren entender qué está sucediendo realmente cuando se ejecuta un agente de IA. No los diagramas de marketing. No las abstracciones del "Motor de Razonamiento". Las peticiones HTTP reales. El `while` loop real.

Si puedes leer Python y has construido una aplicación web o una herramienta de CLI antes, tienes todo lo que necesitas.

Lo Que Vas a Construir

Nanocode es un agente de programación que se ejecuta en tu terminal. Al final de este libro, podrá:

- Leer y escribir archivos en tu base de código
- Ejecutar comandos de shell
- Buscar código usando Python puro
- Recordar contexto entre sesiones
- Solicitar permiso antes de operaciones peligrosas
- Buscar en la web documentación y respuestas

Lo construirás desde cero usando solo `requests`, `subprocess` y `pytest`. Sin LangChain. Sin bases de datos vectoriales. Sin "frameworks de orquestación". Solo Python que puedes depurar con `print()`.

La única excepción: El Capítulo 11 añade `ddgs` para búsqueda web—una única dependencia ligera.

Enfoque de Pruebas

Este libro utiliza un enfoque de "pruebas en paralelo". Para cada funcionalidad:

1. Introducimos el concepto—por qué lo necesitamos
2. Mostramos la prueba primero, para que sepas cómo se ve el éxito
3. Implementamos el código para hacer que la prueba pase
4. Verificamos con pytest

Esto enseña el pensamiento del *Desarrollo Guiado por Pruebas* sin toda la ceremonia de rojo-verde-refactorizar en formato impreso. También resuelve un problema crítico: no puedes probar una aplicación basada en LLM llamando realmente al LLM. Las llamadas a la API son lentas, costosas y no deterministas.

A partir del Capítulo 3, introducimos el patrón `FakeBrain`—un doble de prueba que devuelve respuestas predecibles. Esto te permite ejecutar toda tu suite de pruebas sin hacer una sola llamada a la API ni gastar un solo centavo.

Ejemplos de Código

Este libro sigue un enfoque de "Código Primero". Cada capítulo se construye sobre el anterior, y cada ejemplo de código se extrae de archivos funcionales.

Para obtener el código:

- **GitHub**: Clona o descarga desde GitHub.[1]
- **Leanpub**: El código fuente completo también está incluido en los recursos descargables con tu compra.

El código está organizado por capítulos (`ch01/`, `ch02/`, etc.). Cada carpeta contiene:

- `nanocode.py` — El agente completo y ejecutable para ese capítulo
- `test_nanocode.py` — Pruebas que verifican que el código funciona sin llamadas a la API

Puedes copiar cualquier carpeta de capítulo y comenzar desde allí. Ejecuta `pytest` para verificar que tu código coincide con el comportamiento esperado.

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

Convenciones Utilizadas en Este Libro

A lo largo del libro, verás tres tipos de notas especiales:

> **Consejo de Desarrollo**: Sabiduría arquitectónica que se aplica más allá de este proyecto. Estos son patrones que puedes reutilizar en tu propio trabajo.

Advertencia: Riesgos de seguridad o protección. Presta atención a estos—ignorarlos puede llevar a archivos eliminados o claves API filtradas.

Nota al Margen: Análisis en profundidad y digresiones. Contexto útil, pero puedes saltártelos en una primera lectura sin perder el hilo.

Los análisis de código siguen un patrón consistente: primero el contexto (por qué necesitamos este código), luego el código en sí, y después una explicación línea por línea de las partes importantes.

Es hora de escribir código.

Parte I: El Cerebro

En estos capítulos iniciales, usted construirá el esqueleto de un agente: un bucle `while` que se comunica con Claude a través de HTTP puro. Para el Capítulo 4, habrá implementado soporte para múltiples proveedores LLM mediante el Adapter Pattern, y comprenderá que un "agente de IA" es simplemente un bucle, un cerebro y una lista de mensajes.

Capítulo 1: El Manifiesto Cero Magia

Si has intentado construir una aplicación de IA en los últimos dos años, probablemente hayas sentido la "Fatiga de Frameworks."

Instalas una biblioteca popular. Importas un `ReasoningEngine`. Llamas a `.run()`. Funciona como por arte de magia en el ejemplo "Hello World". Pero en el momento en que intentas hacer algo real —como editar una línea específica en un archivo Python sin eliminar las importaciones— se rompe.

Y como usaste un framework, no puedes arreglarlo. Te quedas atascado excavando a través de capas de clases abstractas, patrones factory y "Chains" tratando de encontrar el único prompt que está causando la alucinación.

No vamos a hacer eso aquí.

Este libro es una rebelión contra la "Magia". Vamos a tomar el enfoque "Cero Magia": construir un agente de programación de nivel producción llamado **Nanocode** en Python puro. Sin LangChain, sin AutoGPT, sin Pydantic.

¿Por qué? Porque un agente autónomo no es magia. Es solo un bucle `while`.

¿Qué es realmente un Agente?

Si dejamos de lado el marketing de capital de riesgo, un "Agente" es simplemente un **termostato**.

Un termostato lee la temperatura (entrada), la compara con el objetivo (decisión) y enciende el calentador (acción). Luego espera y repite. Eso es todo. Un agente de IA hace lo mismo, solo que con texto en lugar de temperatura.

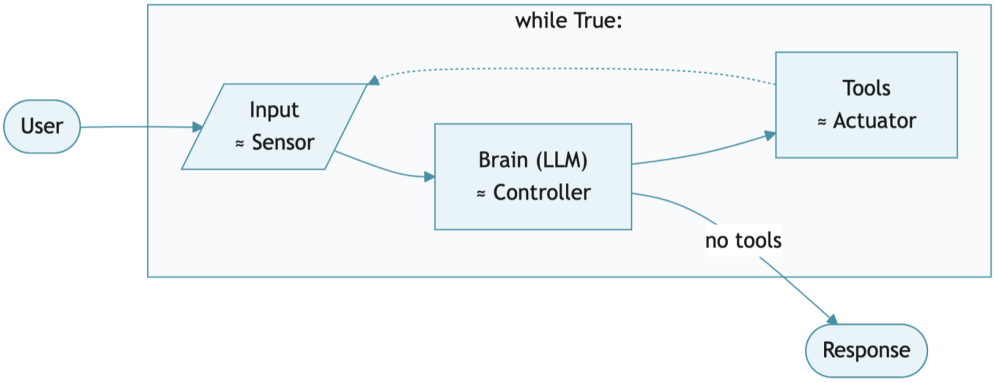


Figura 1. El Bucle del Agente: La entrada del usuario fluye a través de un bucle while donde la Entrada (Sensor) alimenta al Cerebro/LLM (Controlador), que activa las Herramientas (Actuador), volviendo a la Entrada hasta que el Cerebro produce una Respuesta.

Más específicamente, un agente tiene cuatro partes. El *Cerebro* es el LLM —una función sin estado donde envías texto y devuelve texto. Llama a las *Herramientas* —funciones como Leer Archivo y Ejecutar Comando— para interactuar con el mundo exterior. Todo esto se encuentra dentro de un *Bucle* (un `while True`) que sigue ciclando hasta que la tarea está terminada, con la *Memoria* —solo una lista de Python— acumulando el historial de conversación en el camino. (La lista muere cuando el programa termina; añadiremos almacenamiento persistente en el Capítulo 6.)

Si puedes escribir un bucle `while`, puedes construir un agente.

Al construirlo desde cero, tendrás algo que los usuarios de frameworks no tienen: control. Cuando nuestro agente se quede atascado en un bucle, sabrás exactamente qué línea de código lo causó. Cuando la factura de la API se vuelva demasiado alta, verás exactamente dónde se están fugando los tokens.

Qué Vamos a Construir

Nanocode es una herramienta CLI que se ejecuta en tu terminal. Hablas con ella como si fuera un colega. Lee tus archivos. Ejecuta tus comandos. Edita tu código.

Al final de este libro, la habrás conectado a Claude Sonnet 4.6 (o DeepSeek, o un modelo local a través de Ollama). Le darás manos —herramientas para leer archivos, escribir archivos y ejecutar comandos de shell— y ojos para buscar en tu base de código. Y construirás un arnés de seguridad para que no pueda ejecutar accidentalmente `rm -rf /`.

Configuración del Proyecto

> **Consejo de Desarrollo**: La mayor amenaza para un proyecto de IA es el "Deterioro de Dependencias". Las bibliotecas de IA se mueven rápido y rompen cosas. Al limitarnos a `requests` y `subprocess`, es probable que este agente siga funcionando dentro de 5 años.

1. Inicializar el Proyecto

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

2. Crear un Entorno Virtual

Nunca instales herramientas de IA de forma global. Entran en conflicto con los paquetes del sistema.

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

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

3. Instalar Dependencias

Solo necesitamos tres bibliotecas:

- `requests` — Para comunicarse con las APIs de LLM.
- `python-dotenv` — Para cargar las claves API desde un archivo `.env`.
- `pytest` — Para probar nuestro código sin realizar llamadas API.

Crea `requirements.txt`:

```
requests
python-dotenv
pytest
```

Instalación:

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

4. Asegura Tus Claves

Advertencia: Si subes tu clave de API a GitHub, los bots la detectarán y vaciarán tu cuenta en cuestión de minutos.

Crea `.gitignore`:

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

La Excepción AgentStop

Antes de escribir el bucle de eventos, necesitamos un mecanismo de salida limpia. Una excepción es mejor que tener sentencias break dispersas por todo el código.

El Contexto: Las excepciones no son solo para errores; también son un mecanismo de control de flujo. Cuando el usuario escribe `/q`, lanzamos `AgentStop`. El bucle principal la captura y sale limpiamente.

El Código:

```
# --- Exceptions ---

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

Esto va en la parte superior de `nanocode.py`. Es una excepción marcadora—sin lógica, solo una señal.

La Clase Agent

Ahora la abstracción fundamental: la clase `Agent`. Mantiene el estado y la lógica en un solo lugar, lo que facilita las pruebas.

El Contexto: *Podríamos* poner toda la lógica en `main()`. Pero entonces tendríamos que simular `input()` y `print()` para probarla. Al extraer la lógica en `Agent.handle_input()`, podemos probarla directamente.

El Código:

```
class Agent:
    """A coding agent that processes user input."""

    def __init__(self):
        pass

    def handle_input(self, user_input):
        """Handle user input. Returns output string, raises AgentStop to quit."""
        if user_input.strip() == "/q":
            raise AgentStop()

        if not user_input.strip():
            return ""

        return f"You said: {user_input}\n(Agent not yet connected)"
```

La explicación detallada:

- **Líneas 13-14**: Constructor vacío por ahora. Añadiremos `brain` y `tools` en capítulos posteriores.
- **Líneas 18-19**: Verifica el comando de salida `/q`. Lanza `AgentStop` en lugar de devolver un valor especial.

- **Líneas 21-22**: Omite la entrada vacía. Devuelve una cadena vacía (sin salida para mostrar).
- **Línea 24**: Devuelve la entrada como eco. Esto es provisional—más adelante, enviaremos esto al Brain.

Definiendo el éxito con pruebas

Antes de escribir el bucle principal, necesitamos pruebas.

Crea `test_nanocode.py`:

```
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  ")
```

Ejecute las pruebas:

```
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
```

Todo verde. Nuestro agente maneja los casos básicos correctamente.

Aparte: ¿Por qué pytest? Detecta las funciones que comienzan con `test_` y las ejecuta. Sin código repetitivo, sin necesidad de clases. El código de prueba es Python puro, sin magia.

El Bucle Principal

Ahora el envoltorio delgado de I/O que conecta el agente con la terminal:

```
def main():
    agent = Agent()
    print("⚡ Nanocode v0.1 initialized.")
    print("Type '/q' to quit.")

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

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


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

La Explicación Detallada:

- **Líneas 30-32:** Crea el agente e imprime los mensajes de inicio.

- **Línea 36:** `input()` se bloquea y espera a que el usuario escriba algo.
- **Líneas 37-39:** Llama a `handle_input()` e imprime cualquier salida.
- **Líneas 41-43:** Captura `AgentStop` (desde `/q`) o `KeyboardInterrupt` (desde Ctrl+C) y sale limpiamente.

Nota adicional: La función `input()` de Python lee una línea a la vez. Todos los prompts en este libro son de una sola línea. Esto mantiene el código simple—los agentes en producción utilizan métodos de entrada más sofisticados como readline o TUIs completos.

Observa la separación: `Agent.handle_input()` contiene toda la lógica. `main()` es solo el pegamento de E/S. Esto hace que el agente sea comprobable sin necesidad de simular stdin/stdout.

Ejecútalo

```
python nanocode.py
```

Debería ver:

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

 hello
You said: hello
(Agent not yet connected)

 /q

Exiting...
```

Este es el chasis. El motor viene después.

Conclusión

Eso es el chasis: una clase `Agent`, un método `handle_input()`, un bucle `while True`. Todavía no hace nada útil, pero todo lo que construiremos a partir de aquí se conecta a este esqueleto. Las pruebas aseguran que no rompamos lo que ya funciona mientras avanzamos.

Capítulo 2: La Solicitud sin Procesar

La mayoría de los tutoriales te dirán que ejecutes `pip install anthropic`. No vamos a hacer eso.

Los SDK ocultan la verdad. Añaden capas de abstracción que hacen que un "Hola Mundo" sea fácil pero convierten la depuración de un "Error 400" en una pesadilla. Cuando aprendes el SDK, aprendes el SDK. Cuando aprendes la llamada HTTP sin procesar, aprendes el protocolo que está debajo de cada SDK.

Vamos a enviar un mensaje a Claude usando solamente la biblioteca `requests`. Claude es el modelo de lenguaje insignia de Anthropic—uno de los modelos más capaces para tareas de programación.

Obtener una Clave API

Para hablar con Claude, necesitas una Clave API. Esta es una larga cadena de caracteres que actúa como tu tarjeta de crédito.

1. Ve a la Consola de Anthropic.[1]
2. Regístrate y añade un método de pago (crédito mínimo de $5).
3. Crea una nueva Clave API y nómbrala `nanocode`.
4. Copia la clave (comienza con `sk-ant-...`).

Advertencia: Trata esta clave como una contraseña. Cualquiera que la tenga puede gastar tu dinero.

La Bóveda (.env)

Necesitamos un lugar seguro para almacenar esta clave. Nunca ponemos claves directamente en el código.

Crea un archivo llamado `.env` en la raíz de tu proyecto:

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

```
touch .env
```

Ábralo y pegue su clave:

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

Instalamos `python-dotenv` en el Capítulo 1 exactamente para este propósito—lee `.env` y carga los valores en `os.environ`.

La Anatomía de una Solicitud

Para comunicarnos con un LLM, enviamos una solicitud HTTP POST a:

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

Esta solicitud necesita tres elementos: autenticación en los encabezados (tu clave de API), configuración en el cuerpo (qué modelo, cuántos tokens) y el mensaje en sí.

Los Encabezados

Anthropic requiere tres encabezados:

Header	Value	Purpose
`x-api-key`	Tu clave secreta	Autenticación
`anthropic-version`	`2023-06-01`	Versión de API
`content-type`	`application/json`	Formato

La Carga Útil

La “API de Mensajes” espera una lista de diccionarios de mensajes:

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

Cada mensaje tiene un `role` (ya sea `"user"` o `"assistant"`) y `content` (el texto).

El Código

Crea un archivo llamado `test_api.py`. Esta es una "prueba de humo" para comprobar que nuestra conexión funciona. Lo eliminaremos más tarde.

El Contexto: Estamos escribiendo código lineal y procedimental. Sin funciones, sin clases. Queremos ver el nivel más básico.

El Código:

```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-4-6",
    "max_tokens": 4096,
    "messages": [
        {"role": "user", "content": "Hello, are you ready to code?"}
```

```
    ]
}

# 5. Fire! (No safety net)
print("📡 Sending request to Claude...")
response = requests.post(url, headers=headers, json=payload, timeout=120)

# 6. Inspect the raw result
print(f"Status: {response.status_code}")

if response.status_code == 200:
    # Success: Print the beautiful JSON
    print("Response:")
    print(json.dumps(response.json(), indent=2))
else:
    # Failure: Print the ugly raw text so we can debug
    print("Error:", response.text)
```

La Explicación Detallada:

- **Línea 7:** `load_dotenv()` encuentra el archivo `.env` y carga las variables en `os.environ`.
- **Línea 8:** Obtenemos la clave API. Nunca la codifiques directamente en el código.
- **Líneas 11-13:** Validación básica. Sin esto, una clave faltante causa un error confuso de `NoneType` en el diccionario de headers.
- **Línea 21:** El header `anthropic-version` es obligatorio. Si lo omites, la API te rechazará.
- **Línea 27:** `claude-sonnet-4-6` especifica qué modelo queremos.
- **Línea 28:** `max_tokens` es requerido. Limita la longitud de la respuesta y previene costos descontrolados.
- **Línea 36:** Enviamos la petición con un timeout de 2 minutos. Sin try/except—si la red está caída, dejamos que Python falle. Necesitas ver dónde falla.
- **Líneas 41-47:** Verificamos el código de estado. `200` significa éxito (impresión formateada del JSON). Cualquier otro código significa que imprimimos el texto del error en bruto para depuración.

Ejecútalo

```
python test_api.py
```

Si todo funciona, debería ver:

```
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
  }
}
```

Resolución de problemas

Error	Causa	Solución
401 Unauthorized	Clave API incorrecta	Verifica que `.env` se está cargando. Imprime `os.environ.get("ANTHROPIC_API_KEY")` para verificar.
400 Bad Request	JSON mal formado	¿Olvidaste `max_tokens`? ¿Es `messages` una lista?
429 Rate Limit	Demasiadas solicitudes o sin créditos	Espera, o añade créditos a tu cuenta.

Limpieza

Probamos que podemos hablar con el cerebro. Elimina `test_api.py`: ya no lo necesitamos.

Nota al margen: Este `test_api.py` es código desechable, una prueba de humo de un solo uso. Las pruebas automatizadas reales (con `FakeBrain` y pytest) llegan en el Capítulo 3. Siempre deberías eliminar este archivo después de verificar que tu conexión API funciona.

Consejo de desarrollo: Cada llamada a la API cuesta dinero. Los tokens de entrada (lo que envías) son baratos. Los tokens de salida (lo que el modelo escribe) son aproximadamente 5 veces más caros. Mantén tus prompts concisos.

Nota al margen: Para monitorear tu gasto, revisa la pestaña de Uso en la Consola de Anthropic. A principios de 2026, una sesión típica de programación con 20-30 intercambios cuesta entre $0.10-$0.50 con Claude Sonnet. El campo `usage` en la parte inferior del JSON de respuesta muestra el conteo exacto de tokens; podrías registrar estos datos para hacer un seguimiento programático de los costos.

Conclusión

Esa es una llamada API sin procesar: encabezados, carga útil JSON, análisis de respuesta. No hay abstracción entre tú y el nivel bajo. Cuando algo se rompa (y se romperá), sabrás exactamente qué capa falló porque solo hay una capa.

Un problema: Claude tiene amnesia total. Cada solicitud es una pizarra en blanco. Vamos a simular la memoria reproduciendo todo el historial de conversación en cada turno.

Capítulo 3: El Bucle Infinito

Tenemos un problema.

Ejecuta el script del Capítulo 2 dos veces. Di "Me llamo Alice". Claude te saluda. Ejecútalo de nuevo y pregunta "¿Cuál es mi nombre?" Claude responde "No lo sé".

Esto ocurre porque los LLM son *sin estado*. Tienen amnesia total. Cada solicitud es como si fuera la primera vez que te conocen.

Para construir un agente, necesitamos arreglar esto creando una memoria artificial.

La Ilusión de la Memoria

La "memoria" en un LLM no es un disco duro. Es un archivo de registro.

Cuando chateas con ChatGPT, no "recuerda" lo que dijiste hace 5 minutos. Entre bastidores, el código envía el **historial completo de la conversación** al modelo con cada nuevo mensaje.

Figura 2. Acumulación de contexto: El Turno 1 envía solo "Usuario: Hola" a la API. El Turno 2 envía el historial completo—"Usuario: Hola", "Asistente: Hola", "Usuario: ¿Cómo estás?"—a la API.

El modelo ve la transcripción completa cada vez. Ese es el truco.

Vamos a implementar este *bucle de contexto* manualmente. Pero primero, necesitamos hacer que nuestro código sea comprobable.

El Problema de las Pruebas

Aquí hay una verdad difícil: no puedes probar una aplicación basada en LLM haciendo llamadas reales al LLM.

Las llamadas a la API son lentas (2-10 segundos cada una), costosas (dinero real por llamada) y no deterministas (podrías obtener una respuesta diferente cada vez). Imagina ejecutar una suite de pruebas que cuesta $5 y tarda 20 minutos. Nunca la ejecutarías.

La solución es la *inyección de dependencias*. En lugar de hacer una codificación estática de la llamada a la API dentro de nuestro agente, pasamos un objeto "cerebro". En producción, el cerebro es Claude. En las pruebas, el cerebro es una simulación que devuelve respuestas predecibles.

Estableceremos este patrón ahora, antes de escribir más código de producción.

Tipos de Respuesta

Antes de construir el cerebro, necesitamos definir lo que devuelve. La API de Claude envía JSON complejo con múltiples bloques de contenido. Necesitamos objetos Python simples con los que trabajar.

El Contexto: Claude puede devolver texto, llamadas a herramientas o ambos en una sola respuesta. Necesitamos clases de datos para representar estas posibilidades.

El Código:

```
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` representa al cerebro pidiéndonos ejecutar una herramienta. El `id` es un identificador único para el seguimiento (Claude lo necesita cuando le informamos los resultados). El `name` es qué herramienta ejecutar. El `args` es un diccionario de parámetros.

No usaremos `ToolCall` todavía—el cerebro no puede llamar a herramientas—pero lo definimos ahora porque es parte del tipo de respuesta `Thought`. Cuando agreguemos herramientas, Claude devolverá estos cuando quiera leer un archivo o ejecutar un comando.

```
26  class Thought:
27      """Standardized response from any Brain."""
28
29      def __init__(self, text=None, tool_calls=None):
30          self.text = text  # str or None
31          self.tool_calls = tool_calls or []  # list of ToolCall
```

Un `Thought` es lo que el cerebro devuelve después de pensar. Puede contener texto, llamadas a herramientas, ambos o ninguno. Esta abstracción nos permitirá reemplazar Claude por DeepSeek más adelante sin cambiar ningún otro código.

El Patrón del Cerebro Falso

Ahora podemos construir un cerebro falso para realizar pruebas.

El Contexto: Necesitamos un cerebro que devuelva respuestas predecibles, rastree cuántas veces fue llamado y registre qué conversación recibió.

El Código:

```
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")
```

Esto va en `test_nanocode.py`, no en el código de producción. Observa que `FakeBrain` tiene la misma interfaz que tendrá nuestro cerebro real—un método `think()` que toma una conversación y devuelve un `Thought`.

Nota al margen: Este patrón—reemplazar una dependencia real con una simulación predecible para pruebas—se llama inyección de dependencias. El artículo de Martin Fowler "Mocks Aren't Stubs"[1] explica las variaciones (fakes, stubs, mocks, spies). Para las pruebas de LLM, normalmente todo lo que necesitas es un fake simple con respuestas predefinidas.

Definiendo el Éxito

Antes de escribir el código de producción, definamos cómo se ve el éxito. Estas pruebas guiarán nuestra implementación.

Prueba 1: El cerebro devuelve una respuesta

```
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!"
```

Observe que pasamos `brain=brain` al Agent. Esto es la inyección de dependencias en acción.

Prueba 2: La conversación se acumula

```
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
```

Después de cada intercambio, la conversación debe contener tanto el mensaje del usuario como la respuesta del asistente.

Prueba 3: Estructura correcta del mensaje

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

```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"
```

Los mensajes deben tener el formato exacto que Claude espera: `{"role": "user", "content": "..."}`.

Test 4: Brain recibe la conversación

```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"
```

El cerebro debe recibir la conversación completa, no solo el mensaje actual.

Ejecuta estas pruebas ahora—todas deberían fallar:

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

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

Bien. Ahora hagamos que pasen.

La Clase Claude

Ahora el verdadero cerebro.

El Contexto: Necesitamos una clase que envuelve la API de Claude. Debe manejar la autenticación, enviar el historial de conversación y analizar la respuesta para convertirla en un `Thought`.

El Código:

```
36  class Claude:
37      """Claude API - the brain of our agent."""
38
39      def __init__(self):
40          self.api_key = os.getenv("ANTHROPIC_API_KEY")
41          if not self.api_key:
42              raise ValueError("ANTHROPIC_API_KEY not found in .env")
43          self.model = "claude-sonnet-4-6"
44          self.url = "https://api.anthropic.com/v1/messages"
45
46      def think(self, conversation):
47          headers = {
48              "x-api-key": self.api_key,
49              "anthropic-version": "2023-06-01",
50              "content-type": "application/json"
51          }
52          payload = {
53              "model": self.model,
54              "max_tokens": 4096,
55              "messages": conversation
56          }
57
58          print("(Claude is thinking...)")
59          response = requests.post(self.url, headers=headers, json=payload, timeout=120)
60          response.raise_for_status()
61          return self._parse_response(response.json()["content"])
```

La Explicación Detallada:

- **Líneas 39-41:** Carga la clave API y falla rápidamente si no está presente.
- **Líneas 43-44:** Almacena la configuración. Haremos el modelo configurable más adelante.
- **Línea 46:** El método `think()` es la interfaz del cerebro—igual que `FakeBrain`.
- **Líneas 52-55:** La carga útil incluye `"messages": conversation`—el historial completo, no solo el mensaje actual. Este es el bucle de contexto.

- **Línea 61**: Analiza el formato de respuesta complejo de Claude para convertirlo en nuestro `Thought` simple.

Ahora el analizador de respuesta:

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

La API de Claude devuelve una lista de "bloques de contenido". Cada bloque tiene un `type` —ya sea `"text"` o `"tool_use"`. Recopilamos todos los bloques de texto en una única cadena y convertimos los bloques tool_use en objetos `ToolCall`.

La Clase Agent (Actualizada)

Ahora actualizamos el `Agent` del Capítulo 1 para aceptar un cerebro y mantener un historial de conversación.

El Código:

```
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}"
```

La explicación detallada:

- **Líneas 89-91:** Acepta un cerebro mediante inyección de dependencias. Inicializa una lista de conversación vacía.
- **Línea 101:** Añade el mensaje del usuario al historial *antes* de llamar al cerebro.
- **Líneas 103-107:** Llama al cerebro, extrae el texto, añade la respuesta al historial.
- **Líneas 108-110:** Si la llamada a la API falla, elimina el mensaje del usuario que acabamos de añadir. Esto mantiene la conversación en un estado válido.

Presta atención a la línea 101: añadimos el mensaje del usuario *antes* de llamar al cerebro. El cerebro necesita ver la conversación completa, incluyendo el mensaje actual.

> **Consejo para desarrolladores:** Cuando las cosas van mal, tu primera herramienta de depuración es `print(self.conversation)`. Esto muestra exactamente lo que el cerebro está viendo. Los mensajes mal formados, roles faltantes o contenido truncado se vuelven

obvios cuando inspeccionas la lista en bruto.

El bucle principal (Actualizado)

El bucle principal es ahora solo un envoltorio de E/S ligero:

```
115 def main():
116     brain = Claude()
117     agent = Agent(brain)
118     print("⚡ Nanocode v0.2 (Conversation Memory)")
119     print("Type '/q' to quit.\n")
120 
121     while True:
122         try:
123             user_input = input("□ ")
124             output = agent.handle_input(user_input)
125             if output:
126                 print(f"\n{output}\n")
127 
128         except (AgentStop, KeyboardInterrupt):
129             print("\nExiting...")
130             break
131 
132 
133 if __name__ == "__main__":
134     main()
```

Toda la lógica está en la clase `Agent`. El bucle simplemente lee la entrada, llama a `handle_input()`, e imprime el resultado. Esta separación hace que el agente sea verificable—podemos probar `Agent.handle_input()` directamente sin necesidad de simular `input()` o `print()`.

Verificar que las Pruebas Pasen

Ejecuta las pruebas nuevamente:

```
1 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
```

Todo en verde. Las pruebas verifican nuestra implementación sin realizar una sola llamada a la API.

Probar la Memoria

Ahora probemos con el cerebro real:

```
python nanocode.py
```

Pruebe esta conversación:

```
 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.
```

La lista de conversación está cumpliendo su función.

El Problema de la Ventana de Contexto

Quizás estés pensando: “¿Puedo mantener esto funcionando para siempre?”

No.

En cada iteración del bucle, la lista `messages` crece:

Turno	Tokens Aproximados
1	50
10	5.000
100	50.000

Eventualmente, alcanzas el *límite de contexto*—200k tokens para Claude Sonnet, 128k para DeepSeek, y tan bajo como 4k para algunos modelos locales. Si lo excedes, la API devuelve `400 Bad Request`. Nuestro manejo de errores en la línea 108 captura esto y reporta el error, por lo que el agente no fallará silenciosamente. Pero la conversación está efectivamente bloqueada—cada mensaje subsiguiente también fallará, ya que el historial sigue siendo demasiado largo.

Por ahora, reiniciar el agente limpia el historial y te permite volver a empezar. Añadiremos una apropiada *compactación de contexto*—rastreando el uso de tokens desde la respuesta de la API y resumiendo automáticamente los mensajes antiguos antes de que desborden—cuando construyamos el bucle de retroalimentación en el Capítulo 9. Ahí es donde las conversaciones realmente se descontrolan, y donde la solución demostrará su valor.

> **Consejo para Desarrolladores:** ¿Notas el `print("(Claude is thinking...)")`? Las llamadas de red toman entre 2 y 10 segundos. Siempre proporciona retroalimentación inmediata de que la tecla Enter funcionó, o los usuarios presionarán Ctrl+C pensando que se congeló.

Conclusión

Claude ahora recuerda—o mejor dicho, lo hemos engañado para que piense que lo hace. La lista de conversación crece con cada turno, y `FakeBrain` nos permite probar todo sin gastar un centavo.

Ambos patrones se mantendrán a lo largo del resto del libro. Cada cerebro que construyamos (Claude, DeepSeek, Ollama) implementará la misma interfaz `think()`, y `FakeBrain` los probará a todos.

Un cabo suelto: nuestro código está directamente vinculado a la API de Anthropic. Si queremos añadir DeepSeek o un modelo local, tendríamos que duplicar mucho código.

Capítulo 4: El Adaptador Universal

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

El Patrón Adaptador

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Resiliencia HTTP

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

La Interfaz Brain

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

El FakeBrain (Actualizado)

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

El Cerebro Claude (Refactorizado)

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

El Cerebro DeepSeek

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

El Registro de BRAINS

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

La Clase Agent (Actualizada)

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Pruebas para Soporte Multi-Cerebro

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

El Bucle Principal (Actualizado)

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Configuración de DeepSeek

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Pruébalo

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

“Solo Movimos Código”

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Conclusión

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Parte II: Las Manos

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Capítulo 5: El Protocolo de Herramientas

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Cómo Funcionan Realmente las Herramientas

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Definiendo la Interfaz de Herramientas

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

La Herramienta ReadFile

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

La Herramienta WriteFile

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Funciones Auxiliares de Herramientas

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Actualizando la Clase Thought

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Actualizando la Clase Claude

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

La Clase Agent con Herramientas

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

El Bucle Principal

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Pruébalo

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Conclusión

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Capítulo 6: El Scratchpad (Memoria)

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

La Memoria "Cero Magia"

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

La Clase Memory

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

La Clase ToolContext

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

La Herramienta SaveMemory

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Actualizando la Clase Claude

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Elaboración del Prompt del Sistema

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Actualizando la Clase Agent

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

El Bucle Principal (Actualizado)

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Prueba de Persistencia

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Conclusiones

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Capítulo 7: El Arnés de Seguridad (Modo Planificación)

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

El Concepto

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Primero las Pruebas

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Extendiendo ToolContext

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

La Herramienta WriteFile Protegida

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

La Clase Agent (Actualizada)

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

El Bucle Principal (Actualizado)

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Probando el Arnés de Pruebas

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

La Psicología del “Plan”

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Conclusión

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Capítulo 8: El Pipeline de Contexto (Mapeo y Búsqueda)

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

La Herramienta ListFiles

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

La Herramienta SearchCodebase

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Actualizar la Lista de herramientas

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

La Prueba “Zoom In”

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Espera, ¿esto es RAG?

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Conclusión

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Capítulo 9: La Prueba de Realidad (Ejecutar Código)

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

El Ciclo de Retroalimentación

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Primero las Pruebas

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

La Herramienta RunCommand

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

La Trampa Interactiva

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

La Demostración de Auto-Reparación

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

El flujo de trabajo de TDD

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

La Edición Quirúrgica

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

El Bucle Cerrado

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Compactación del Contexto

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Consideraciones de Seguridad

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Conclusión

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Parte III: La Frontera

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Capítulo 10: Desconectándose (Modelos locales)

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

El compromiso

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Instalación de Ollama

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

La Clase Cerebro de Ollama

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Ejecutando con Ollama

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

El Experimento del “Bucle Infinito”

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Las Diferencias Prácticas

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

El Flujo de Trabajo Híbrido

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Selección de Modelos

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Resolución de problemas de Ollama

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Conclusión

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Capítulo 11: La Extensión (Búsqueda Web)

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Paso 1: El Meta-Prompt

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Paso 2: La Cirugía

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Paso 3: La Implementación de Referencia

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Paso 4: Las Pruebas

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Automodificación

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Conclusión

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.

Capítulo 12: El Proyecto Final (Construyendo un Juego)

¿Puede Nanocode realmente construir algo?

El "Desafío Cero Código": construir un juego clásico Snake usando Python y Pygame. La regla: no se te permite escribir ni una sola línea de Python. Solo puedes comunicarte con el agente en inglés.

Nota al margen: Esta demostración involucra muchas llamadas a la API. Si alcanzas los límites de frecuencia (HTTP 429), el agente reintentará automáticamente. Para sesiones largas, considera usar un modelo local a través de Ollama para evitar los límites por completo.

Paso 1: Preparación

Desde la raíz de tu proyecto (el directorio que contiene `nanocode.py` y `.env`), crea un directorio de trabajo para el juego:

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

Si está usando el repositorio de código del libro, copie desde ch11 en su lugar:

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

Instalar Pygame:

```
pip install pygame
```

Nota al margen: En Windows, esto debería funcionar sin configuración adicional. En macOS, es posible que necesites instalar primero las bibliotecas SDL: `brew install sdl2 sdl2_image sdl2_mixer sdl2_ttf`. En Linux, instala los paquetes de desarrollo SDL: `sudo apt install libsdl2-dev libsdl2-image-dev libsdl2-mixer-dev libsdl2-ttf-dev`.

Paso 2: El Arquitecto (Modo de Planificación)

Inicia el agente. Comenzamos en modo de planificación porque queremos tener un plano antes de colocar los ladrillos.

```
python nanocode.py
```

El 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.
```

El agente utilizará `write_file` para crear `PLAN.md`. Léelo. Debería describir la clase Snake, la clase Food y el bucle del juego—todo en un solo archivo.

Si parece correcto, apruébalo.

Paso 3: El Constructor (Modo de Acción)

Cambia al modo de acción:

```
/mode act
```

El Prompt:

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

Observa la terminal:

```
  Writing snake.py
```

El agente está generando código basado en el contexto almacenado en `PLAN.md`.

Paso 4: La Verificación de la Realidad

Antes de ejecutar el juego, aumenta el timeout. Abre `nanocode.py` y cambia `timeout=30` a `timeout=300` en `RunCommand.execute()`—los 30 segundos predeterminados no son suficientes para jugar realmente una partida. (Esta es la única excepción al Desafío de Código Cero.)

El Prompt:

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

El agente ejecuta `run_command`. Aparece una ventana. Juegas Snake.

> **Consejo para desarrolladores:** La ventana del juego bloquea al agente. Cuando ejecutas `python snake.py`, el agente espera hasta que cierres la ventana del juego antes de continuar. Esto es normal—el bucle principal de Pygame mantiene el terminal bloqueado.

Si falla: Los LLMs son no determinísticos. Tu agente podría producir un error en el primer intento. Si el juego falla con un error como `AttributeError: 'Snake' object has no attribute 'draw'`, no lo arregles tú mismo. Deja que el agente vea el stderr.

El Prompt:

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

El agente leerá el traceback, usará `read_file` para encontrar el bug, usará `edit_file` para corregirlo, y lo ejecutará de nuevo.

> **Consejo para Desarrolladores**: Este es el ciclo de retroalimentación del Capítulo 9 en acción. El agente escribe, ejecuta, lee el error y corrige. Tú solo observas.

Paso 5: El Giro (Expansión de Características)

El juego funciona, pero es feo. La serpiente son solo cuadrados verdes. Vamos a poner a prueba la capacidad del agente para refactorizar.

El 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.
```

El agente debe:

1. Usar `search_web` para encontrar paletas de colores
2. Usar `read_file` para entender la lógica de renderizado actual
3. Usar `edit_file` para incorporar las nuevas funciones
4. Ejecutar el juego para verificar

Lo Que Sale Mal

Tus resultados serán diferentes a los míos—los LLM no son deterministas. Pero esto es lo que suele suceder, y lo que hay que vigilar.

Fallos comunes en la primera ejecución:

- `ModuleNotFoundError: No module named 'pygame'` — el agente olvidó que necesitas instalarlo, o ejecutó el script en un entorno diferente. Dile que ejecute `pip install pygame` primero.
- `AttributeError` en un método que el agente definió pero escribió mal en el punto de llamada. El agente corrige estos rápidamente una vez que ve el rastreo de error.
- Errores de desplazamiento por uno en la detección de colisiones. La serpiente atraviesa paredes o muere un píxel demasiado pronto. Estos requieren 2-3 iteraciones de editar-ejecutar-corregir.

El arco típico de la sesión:

En mis pruebas, el agente generalmente consigue un juego funcional (pero feo) en 2-4 iteraciones. La primera escritura produce algo que falla. La segunda o tercera corrección lo hace funcionar. El paso de expansión de características (cambios de color, rampas de velocidad) añade otras 3-5 iteraciones mientras el agente lee su propio código, hace ediciones quirúrgicas y verifica cada cambio.

Al final, la conversación tiene 15-20 rondas de profundidad. Si estás usando Claude, observa el disparador de compactación—alrededor de las rondas 12-15 verás "(Compactando conversación...)" cuando el recuento de tokens se acerca al umbral del 75%. Después de la compactación, el agente pierde algunos detalles sobre las primeras rondas pero sigue trabajando. Este es el sistema del Capítulo 9 demostrando su valor.

Donde el agente tiene dificultades:

El sistema de coordenadas y el bucle de eventos de Pygame son complicados. El agente a veces escribe código que se renderiza correctamente pero no maneja adecuadamente la entrada del teclado, o que dibuja la serpiente en el orden incorrecto por lo que la cabeza aparece detrás del cuerpo. Estos son el tipo de errores que un humano detecta instantáneamente pero el agente no puede ver—no tiene retroalimentación visual, solo stdout y stderr. Si el juego se ejecuta sin errores pero se ve mal, necesitarás describir el error visual: "La serpiente se renderiza al revés—la cabeza debería estar al frente."

El punto no es la perfección en el primer intento. Es que el agente *converge*—escribe, ejecuta, lee el error, corrige, repite—usando todas las herramientas que construimos a lo largo de once capítulos.

Conclusión

Planificar, implementar, fallar, depurar, corregir, ejecutar de nuevo—eso es cada capítulo demostrando su valor.

Entonces, ¿hacia dónde va esto desde aquí?

Epílogo

Todo el conjunto son aproximadamente 750 líneas de Python. Sin framework. `nanocode.py` es tuyo. Haz lo que quieras con él:

- Integración con Git que realiza commits automáticamente después de pruebas exitosas

- Depuración basada en capturas de pantalla para trabajo frontend (Claude puede leer imágenes)
- Entrada de voz mediante Whisper para que puedas hablar en lugar de escribir
- MCP para conectar con servicios externos que tu equipo ya utiliza

Los agentes de producción como Claude Code, Cursor y Copilot hacen más que esto—respuestas en streaming, ejecución paralela de herramientas, análisis sintáctico tree-sitter, entornos de ejecución aislados, ventanas de contexto multi-archivo que abarcan miles de archivos. La brecha entre 750 líneas y 750.000 es real. Pero la arquitectura es la misma: un cerebro, un bucle, herramientas, memoria y un arnés de seguridad. Ahora sabes lo que hay detrás de la cortina.

Los modelos seguirán mejorando. El arnés—el bucle, las herramientas, las comprobaciones de seguridad—esa parte es ingeniería. Y esa parte no va a ninguna parte.

Agradecimientos

Este contenido no está disponible en el libro de muestra. El libro se puede comprar en Leanpub en https://leanpub.com/build-your-own-coding-agent-es.