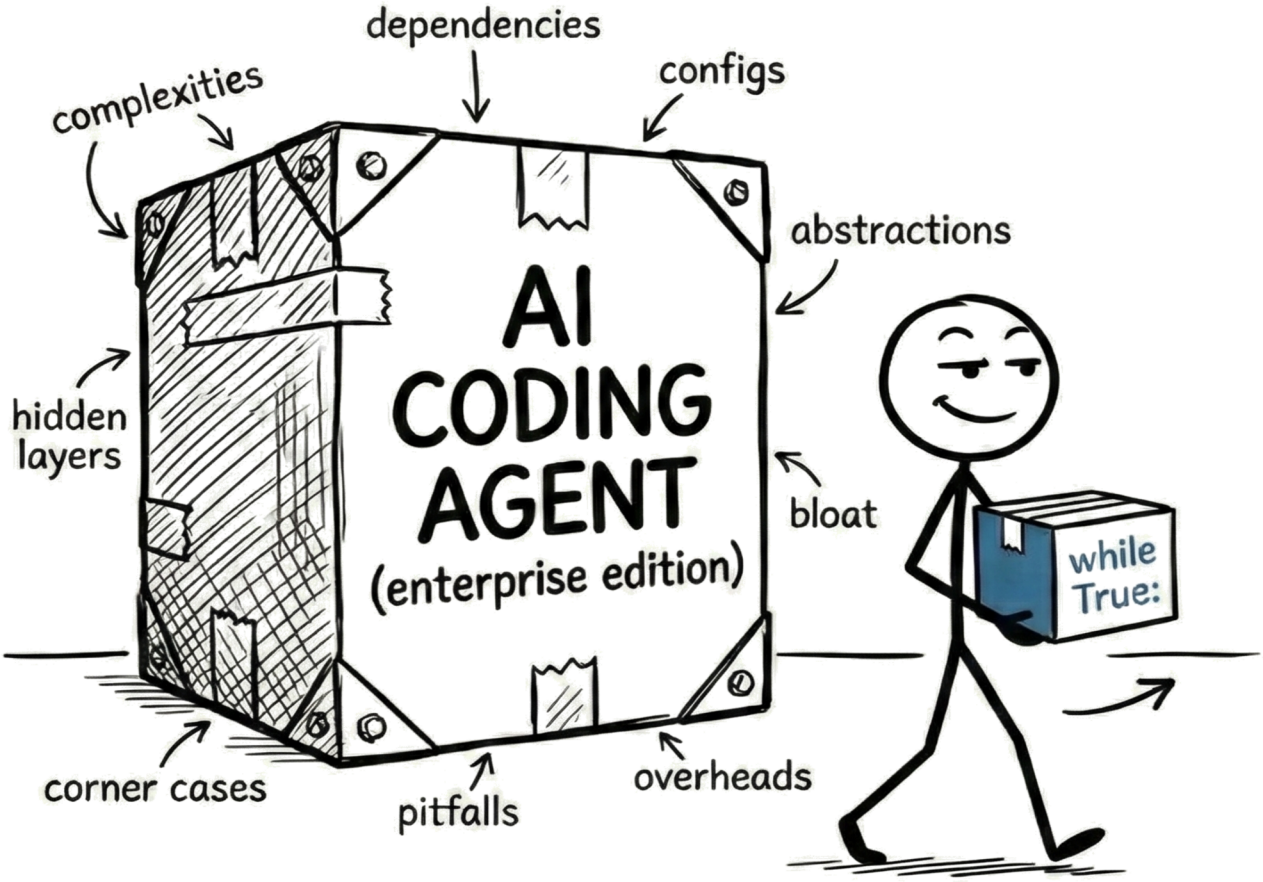
Build Your Own
CODING AGENT

The Zero-Magic Guide to AI Agents in Pure Python



J. Owen

Edición en Español

Construye tu Propio Agente de Programación (Spanish Edition)

Guía Sin-Magia para Agentes de IA en Python Puro

J. Owen

Este libro está a la venta en
https://leanpub.com/build-your-own-coding-agent-es-0bdc30ba

Esta versión se publicó en 2026-04-19

¡Tuitea sobre el libro!

Por favor ayuda a J. Owen hablando sobre el libro en Twitter!

El tuit sugerido para este libro es:

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.

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

Para Quién Es Este Libro

Eres un desarrollador de software escéptico ante el bombo publicitario de la IA.

Has probado los frameworks —y has visto cómo tu aplicación de LangChain alucinaba hasta borrar una base de datos en producción. Y piensas: *"Tiene que haber una forma mejor."*

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

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

Qué Vas a Construir

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

- Leer y escribir archivos en tu base de código
- Ejecutar comandos de shell
- Buscar código usando Python puro
- Recordar el contexto entre sesiones
- Separar la planificación de la ejecución mediante un modo seguro
- Buscar en la web documentación y respuestas

Lo construirás desde cero usando `requests`, `python-dotenv` y `pytest`; el acceso al shell viene del módulo `subprocess` de la biblioteca estándar de Python. 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 la búsqueda web —una única dependencia ligera.

Enfoque de Pruebas

Este libro utiliza un enfoque de "pruebas en paralelo al desarrollo". Para la mayoría de las funcionalidades:

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

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

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

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 se incluye en los recursos descargables con tu compra.

El código está organizado por capítulo (`ch01/`, `ch03/`, `ch04/`, etc.) más una carpeta `appendix/`. La mayoría de las carpetas contienen:

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

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

Dos excepciones: `ch02/` contiene únicamente un script independiente `test_api.py` (usado una vez y descartado), y el snapshot del agente de `ch12/` es funcionalmente igual al de `ch11/`, con artefactos del proyecto final añadidos.

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

Convenciones Utilizadas en Este Libro

A lo largo del libro, verás tres tipos de recuadros destacados:

> **Consejo de Desarrollo**: Sabiduría arquitectónica aplicable más allá de este proyecto. Son patrones que puedes reutilizar en tu propio trabajo.

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

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

Las explicaciones 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 algo de código.

Parte I: El Cerebro

La Parte I construye el esqueleto de un agente: un bucle `while` que se comunica con Claude mediante HTTP puro. Para el Capítulo 4, darás soporte a múltiples proveedores de LLM a través del Patrón Adaptador—y comprenderás que un "agente de IA" no es más que un bucle, un cerebro y una lista de mensajes.

Capítulo 1: El Manifiesto de Cero Magia

Si has intentado construir una aplicación de IA en los últimos dos años, probablemente ya has sentido la *Framework Fatigue*.

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

Y como usaste un framework, no puedes arreglarlo. Te quedas atascado escarbando entre capas de clases abstractas, patrones de factoría 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 adoptar el enfoque de "Cero Magia": construir un agente de codificación de nivel productivo llamado **Nanocode** en Python puro. Sin LangChain, sin AutoGPT, sin Pydantic.

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

¿Qué es realmente un Agente?

Quita el marketing del capital de riesgo, y un "Agente" no es más que 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.

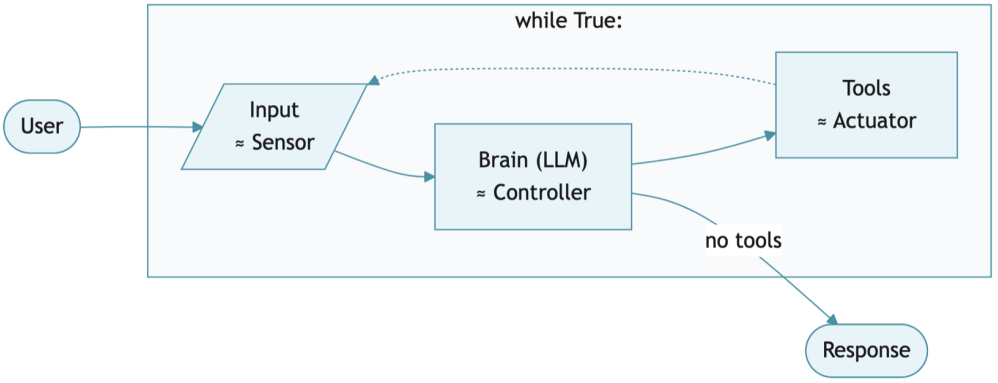


Figura 1. El Bucle del Agente: la entrada del usuario fluye a través de un bucle while donde la Entrada (Sensor) alimenta al Cerebro/LLM (Controlador), que activa las Herramientas (Actuador), volviendo a la Entrada hasta que el Cerebro genera una Respuesta.

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

Si sabes 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 sea demasiado alta, verás exactamente dónde se están filtrando los tokens.

Qué Estamos Construyendo

Nanocode es una herramienta CLI que se ejecuta en tu terminal. Le hablas como a un colega. Lee tus archivos. Ejecuta tus comandos. Edita tu código.

Al final de este libro, lo habrás conectado a Claude Sonnet 4.6 (o DeepSeek, o un modelo local mediante 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 hacer accidentalmente `rm -rf /`.

Configuración del Proyecto

> **Consejo Dev**: La mayor amenaza para un proyecto de IA es la *Dependency Rot*. Las bibliotecas de IA avanzan rápido y rompen cosas. Al ceñirnos 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 de API desde un archivo `.env`.
- `pytest` — Para probar nuestro código sin realizar llamadas a la API.

Crea `requirements.txt`:

```
requests
python-dotenv
pytest
```

Instalar:

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

4. Asegura tus claves

Advertencia: Si subes tu clave de API a GitHub, los bots la extraerán 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 limpio. Una excepción es mejor que tener sentencias `break` dispersas por el código.

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

El código:

```
# --- Exceptions ---

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

Esto va al principio de `nanocode.py`. Es una excepción marcadora—sin lógica, solo una señal.

La Clase Agent

Ahora la abstracción central: la clase `Agent`. Almacena estado y 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)"
```

El recorrido del código:

- **Líneas 13-14:** Constructor vacío por ahora. Añadiremos `brain` y `tools` en capítulos posteriores.
- **Líneas 18-19:** El comando `/q` lanza `AgentStop` en lugar de devolver un valor especial — quien llama decide cómo gestionar el cierre.
- **Línea 24:** Repite la entrada tal cual. Es un marcador de posición — más adelante, enviaremos esto al Brain.

Definir el éxito mediante 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  ")
```

Ejecuta 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 en verde. Nuestro agente maneja correctamente los casos básicos.

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

El Bucle Principal

Ahora la envoltura de E/S ligera que conecta el agente con la 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()
```

El recorrido:

- **Líneas 30-32:** Crea el agente e imprime los mensajes de inicio.
- **Línea 36:** `input()` bloquea la ejecución 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 del bucle.

Aparte: La función `input()` de Python lee una línea a la vez. Todos los prompts de este libro son de una sola línea. Esto mantiene el código simple — los agentes en producción usan métodos de entrada más ricos, como readline o TUIs completas.

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

Ejecútalo

```
python nanocode.py
```

Deberías 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 a continuación.

Concluyendo

Eso es el chasis: una clase `Agent`, un método `handle_input()`, un bucle `while True`. Todavía no hace nada útil—pero todo lo que construyamos a partir de aquí encaja en este esqueleto. Las pruebas garantizan que no rompamos lo que ya funciona conforme 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 SDKs ocultan la verdad. Añaden capas de abstracción que hacen que "Hello World" sea sencillo, pero que depurar un "Error 400" sea una pesadilla. Cuando aprendes el SDK, aprendes el SDK. Cuando aprendes la llamada HTTP en crudo, aprendes el protocolo que subyace a todos los SDKs.

Vamos a enviar un mensaje a Claude usando únicamente la biblioteca `requests`. Claude es el LLM insignia de Anthropic, uno de los modelos más capaces para tareas de programación.

Obtén una Clave de API

Para comunicarte con Claude, necesitas una clave de API. Es una cadena larga de caracteres que funciona como una contraseña vinculada a tu cuenta de facturación.

1. Ve a la Consola de Anthropic.[1]
2. Regístrate y añade un método de pago (mínimo $5 de crédito).
3. Crea una nueva clave de API y nómbrala `nanocode`.
4. Copia la clave (empieza 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 guardar esta clave. Nunca ponemos las 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
```

Ábrelo y pega tu clave:

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

Instalamos `python-dotenv` en el Capítulo 1 precisamente para este propósito: lee el archivo `.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 cosas: 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:

Encabezado	Valor	Propósito
`x-api-key`	Tu clave secreta	Autenticación
`anthropic-version`	`2023-06-01`	Versión de la API
`content-type`	`application/json`	Formato

La Carga Útil

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

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

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

El Código

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

El Contexto: Estamos escribiendo código lineal y procedimental. Sin funciones, sin clases. Queremos ver el metal desnudo.

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)
```

El recorrido:

- **Línea 7:** `load_dotenv()` encuentra el archivo `.env` y carga las variables en `os.environ`.
- **Línea 8:** Obtenemos la clave de API. Nunca la codifiques directamente en el código.
- **Líneas 11-13:** Comprobación básica de validez. Sin esto, una clave faltante provoca un confuso error `NoneType` en el diccionario de encabezados.
- **Línea 21:** El encabezado `anthropic-version` es obligatorio. Si lo omites, la API te rechaza.
- **Línea 27:** `claude-sonnet-4-6` especifica el modelo que queremos.
- **Línea 28:** `max_tokens` es obligatorio. Limita la longitud de la respuesta y evita costos descontrolados.
- **Línea 36:** Enviamos la solicitud con un tiempo de espera de 2 minutos. Sin `try/except` — si la red está caída, deja que Python falle. Necesitas ver dónde ocurre el error.
- **Líneas 41-47:** Verificamos el código de estado. `200` significa éxito (imprime el JSON con formato legible). Cualquier otro valor imprime el texto de error sin procesar para depuración.

Ejecútalo

```
python test_api.py
```

Si todo funciona, deberías 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
  }
}
```

Solución de problemas

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

Limpieza

Hemos demostrado que podemos hablar con el cerebro. Elimina `test_api.py`—hemos terminado con él.

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

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

Aparte: Para monitorear tu gasto, revisa la pestaña de uso en la Anthropic Console. A principios de 2026, una sesión de programación típica con 20 o 30 intercambios cuesta entre $0.10 y $0.50 con Claude Sonnet. El campo `usage` al final del JSON de respuesta muestra el conteo exacto de tokens—podrías registrarlos para rastrear los costos de forma programática.

Conclusión

Eso es una llamada a la API en bruto: encabezados, payload JSON, procesamiento de la respuesta. Sin ninguna abstracción entre tú y el cable. Cuando algo falle (y fallará), sabrás exactamente en qué capa ocurrió porque solo hay una capa.

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

Capítulo 3: El Bucle Infinito

Tenemos un problema.

Imagina ejecutar el script del Capítulo 2 dos veces. Dices "Mi nombre es Alice." Claude te saluda. Lo ejecutas de nuevo y preguntas "¿Cuál es mi nombre?" Claude dice "No lo sé."

Esto se debe a que los LLMs son *sin estado*. Tienen amnesia total. Cada solicitud es la primera vez que se han conocido.

Para construir un agente, necesitamos solucionar esto creando 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.

Implementemos este *bucle de contexto* manualmente. Pero primero, necesitamos que nuestro código sea testeable.

El Problema de las Pruebas

Aquí va una verdad incómoda: no puedes probar una aplicación impulsada por un LLM llamando realmente al LLM.

Las llamadas a la API son lentas (de 2 a 10 segundos cada una), costosas (dinero real por llamada) y no deterministas (puedes obtener una respuesta diferente cada vez). Imagina

ejecutar un conjunto de pruebas que cuesta 5 dólares y tarda 20 minutos. Nunca lo ejecutarías.

La solución es la *inyección de dependencias*. En lugar de codificar en duro 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 un objeto simulado 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 qué 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 objetos de datos simples para representar estas posibilidades. (Omitimos `@dataclass` deliberadamente: estas clases son lo suficientemente simples como para que el decorador ahorre unas pocas líneas a costa de ocultar lo que `__init__` hace realmente.)

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 que ejecutemos una herramienta. El `id` es un identificador único para el seguimiento (Claude lo necesita cuando le devolvemos los resultados). El `name` indica qué herramienta ejecutar. El `args` es un diccionario de parámetros.

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

```
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
```

Un `Thought` es lo que el cerebro devuelve después de pensar. Puede contener texto, llamadas a herramientas, ambos, o ninguno. El campo `thinking` captura el resumen del razonamiento del modelo — veremos de dónde viene eso cuando construyamos la clase Claude más adelante. Esta abstracción nos permitirá reemplazar Claude por DeepSeek en el futuro sin modificar ningún otro código.

El patrón FakeBrain

Ahora podemos construir un cerebro falso para las pruebas.

El contexto: Necesitamos un cerebro que devuelva respuestas predecibles, registre cuántas veces fue llamado y registre la conversación que 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 recibe una conversación y devuelve un `Thought`.

Aparte: Este patrón —reemplazar una dependencia real con un falso predecible para las pruebas— se llama *doble de prueba*. El artículo de Martin Fowler "Mocks Aren't Stubs"[1] explica las variaciones (fakes, stubs, mocks, spies). Para las pruebas de LLM, un fake simple con respuestas prefabricadas suele ser todo lo que necesitas.

Definición del 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!"
```

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

Test 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 de mensaje correcta

[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 viene el verdadero cerebro.

El Contexto: Necesitamos una clase que envuelva la API de Claude. Debe gestionar la autenticación, enviar el historial de conversación y convertir la respuesta en un `Thought`. También habilitamos el *pensamiento extendido*: una característica por la que el modelo escribe apuntes internos antes de responder. Imagínalo como el modelo hablándose a sí mismo en un bloc de notas antes de hablar. Cuesta tokens adicionales, pero la mejora en calidad es significativa, especialmente cuando añadamos herramientas en el Capítulo 5, donde el modelo necesita razonar sobre qué herramienta usar y por qué.

El Código:

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

    def __init__(self):
        self.api_key = os.getenv("ANTHROPIC_API_KEY")
        if not self.api_key:
            raise ValueError("ANTHROPIC_API_KEY not found in .env")
        self.model = "claude-sonnet-4-6"
        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": "enabled",
                "budget_tokens": 10000
            },
            "messages": conversation
        }

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

El Recorrido:

- **Líneas 41-43**: Carga la clave de API y falla de inmediato si no está presente.
- **Líneas 44-45**: Almacena la configuración. Más adelante haremos que el modelo sea configurable.
- **Línea 47**: El método `think()` es la interfaz del cerebro—igual que en `FakeBrain`.
- **Líneas 55-59**: Habilitamos el *extended thinking*—el modelo produce un resumen de razonamiento antes de responder, lo que mejora la calidad en tareas complejas a costa de más tokens. `budget_tokens` limita la cantidad de tokens que el modelo puede gastar en razonamiento (10.000 aquí)—esos tokens cuentan en tu factura igual que los tokens de salida. En nuestra configuración, `max_tokens` cubre el total de la salida, incluyendo tanto el razonamiento como la respuesta, por lo que Anthropic requiere que supere a `budget_tokens`. Con 10.000 tokens de razonamiento y 16.000 de máximo, la respuesta en sí puede usar hasta 6.000 tokens.
- **Línea 60**: El payload incluye `"messages": conversation`—el historial completo, no solo el mensaje actual. Este es el bucle de contexto.
- **Línea 65**: Analiza el complejo formato de respuesta de Claude y lo convierte en nuestro sencillo `Thought`.

Ahora el analizador de respuestas:

```
67      def _parse_response(self, content):
68          """Convert Claude's response format to Thought."""
69          text_parts = []
70          tool_calls = []
71          thinking = None
72
73          for block in content:
74              if block["type"] == "thinking":
75                  thinking = block["thinking"]
76              elif block["type"] == "text":
77                  text_parts.append(block["text"])
78              elif block["type"] == "tool_use":
79                  tool_calls.append(ToolCall(
80                      id=block["id"],
81                      name=block["name"],
82                      args=block["input"]
83                  ))
84
85          return Thought(
86              text="\n".join(text_parts) if text_parts else None,
87              tool_calls=tool_calls,
88              thinking=thinking
89          )
```

La API de Claude devuelve una lista de "bloques de contenido". Cada bloque tiene un `type`—`"thinking"`, `"text"` o `"tool_use"`. El bloque de pensamiento llega primero y contiene un resumen del razonamiento del modelo; lo almacenamos en el `Thought` para que quien lo invoque pueda mostrarlo. Los bloques de texto se convierten en la respuesta, y los bloques `tool_use` se convierten en objetos `ToolCall`. El parser no imprime nada; simplemente convierte el JSON sin procesar en un `Thought` limpio.

La Clase Agent (Actualizada)

Ahora actualizamos el `Agent` del Capítulo 1 para que acepte un cerebro y mantenga el historial de conversación.

El Código:

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

El recorrido:

- **Líneas 97-99**: Acepta un cerebro mediante inyección de dependencias. Inicializa una lista de conversación vacía.
- **Línea 109**: Agrega el mensaje del usuario al historial *antes* de llamar al cerebro.
- **Líneas 112-120**: Llama al cerebro, muestra hasta cinco líneas de pensamiento en texto atenuado (`\033[2m` es el código de escape ANSI para atenuar, `\033[0m` lo restablece), extrae la respuesta y la agrega al historial.
- **Líneas 121-123**: Si la llamada a la API falla, elimina el mensaje del usuario que acabamos de agregar. Esto mantiene la conversación en un estado válido.

Presta atención a la línea 109: agregamos el mensaje del usuario *antes* de llamar al cerebro. El cerebro necesita ver la conversación completa, incluido el mensaje actual.

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

El Bucle Principal (Actualizado)

El bucle principal ahora es simplemente una capa delgada de E/S:

```
128 def main():
129     brain = Claude()
130     agent = Agent(brain)
131     print("⚡ Nanocode v0.2 (Conversation Memory)")
132     print("Type '/q' to quit.\n")
133 
134     while True:
135         try:
136             user_input = input("❯ ")
137             output = agent.handle_input(user_input)
138             if output:
139                 print(f"\n{output}\n")
140 
141         except (AgentStop, KeyboardInterrupt):
142             print("\nExiting...")
143             break
144 
```

```

if __name__ == "__main__":
    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 testeable: probamos `Agent.handle_input()` directamente sin necesidad de simular `input()` ni `print()`.

Verifica que las pruebas pasen

Ejecuta las pruebas de nuevo:

```
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 prueba con el cerebro real:

```
python nanocode.py
```

Prueba esta conversación:

```
 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.
```

La lista de conversación está haciendo su trabajo.

El Problema de la Ventana de Contexto

Quizás estés pensando: "¿Puedo mantener esto corriendo para siempre?"

No.

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

Turno	Tokens Aproximados
1	50
10	5.000
100	50.000

En algún momento, alcanzas el *límite de contexto*—200k tokens para Claude Sonnet, 128k para DeepSeek, y tan solo 4k en algunos modelos locales. Si lo superas, la API devuelve `400 Bad Request`. Nuestro manejo de errores en la línea 121 captura esto e informa el error, así que el agente no fallará en silencio. Pero la conversación queda efectivamente bloqueada—cada mensaje posterior también fallará, ya que el historial sigue siendo demasiado largo.

Por ahora, reiniciar el agente borra el historial y te permite retomar el hilo. Añadiremos la *compactación de contexto* adecuada—rastreando el uso de tokens desde la respuesta de la API y resumiendo automáticamente los mensajes antiguos antes de que se desborden—cuando construyamos el bucle de retroalimentación en el Capítulo 9. Ahí es donde las conversaciones realmente explotan, y donde la solución se ganará su lugar.

> **Consejo de Desarrollo**: Las llamadas de red tardan entre 2 y 10 segundos. El indicador de pensamiento en las líneas 113-117 le da a los usuarios una respuesta visual inmediata de que el agente está trabajando. Sin él, se quedarían mirando un prompt congelado y recurrirían a Ctrl+C.

Conclusión

Claude ahora recuerda—o más bien, lo hemos engañado para que crea que recuerda. La lista de conversación crece con cada turno, y `FakeBrain` nos permite probar todo el sistema 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 cableado directamente a la API de Anthropic. Si quisiéramos añadir DeepSeek o un modelo local, tendríamos que duplicar una gran cantidad de 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-0bdc30ba.

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-0bdc30ba.

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-0bdc30ba.

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-0bdc30ba.

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-0bdc30ba.

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-0bdc30ba.

El Brain 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-0bdc30ba.

El Registro 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-0bdc30ba.

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-0bdc30ba.

Tests para el Soporte de Múltiples Cerebros

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-0bdc30ba.

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-0bdc30ba.

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-0bdc30ba.

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-0bdc30ba.

“Solo Reubicamos el 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-0bdc30ba.

Para Cerrar

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-0bdc30ba.

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-0bdc30ba.

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-0bdc30ba.

Cómo Funcionan las Herramientas en la Práctica

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-0bdc30ba.

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-0bdc30ba.

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-0bdc30ba.

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-0bdc30ba.

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-0bdc30ba.

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-0bdc30ba.

Actualización de 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-0bdc30ba.

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-0bdc30ba.

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-0bdc30ba.

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-0bdc30ba.

Resumiendo

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-0bdc30ba.

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-0bdc30ba.

La Memoria "Sin 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-0bdc30ba.

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-0bdc30ba.

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-0bdc30ba.

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-0bdc30ba.

Actualización de 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-0bdc30ba.

Diseñando el System 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-0bdc30ba.

Actualización de 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-0bdc30ba.

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-0bdc30ba.

Probando la 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-0bdc30ba.

Para terminar

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-0bdc30ba.

Capítulo 7: El Arnés de Seguridad (Plan Mode)

El Concepto

Primero las Pruebas

La Herramienta WritePlan

Una Lista, Dos Vistas

Indicarle al Cerebro en Qué Modo Está

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-0bdc30ba.

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-0bdc30ba.

Probando el Entorno 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-0bdc30ba.

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-0bdc30ba.

Resumiendo

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-0bdc30ba.

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

La Herramienta ListFiles

La Herramienta SearchCodebase

Actualizar la lista de herramientas

La prueba del «Zoom In»

Espera, ¿esto es RAG?

Conclusión

Capítulo 9: La Prueba de Realidad (Ejecutar Código)

El Bucle de Retroalimentación

Primero las Pruebas

La Herramienta RunCommand

La Trampa Interactiva

La Demo de Autoreparación

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-0bdc30ba.

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-0bdc30ba.

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-0bdc30ba.

Reforzando el Bucle

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-0bdc30ba.

Compactación de 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-0bdc30ba.

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-0bdc30ba.

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-0bdc30ba.

Parte III: La Frontera

Capítulo 10: Trabajar sin Conexión (Modelos Locales)

El compromiso

Instalando Ollama

La Clase Brain de Ollama

Ejecutar con Ollama

El Experimento del «Bucle Infinito»

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-0bdc30ba.

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-0bdc30ba.

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-0bdc30ba.

Solución de problemas 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-0bdc30ba.

Cierre

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-0bdc30ba.

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-0bdc30ba.

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-0bdc30ba.

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-0bdc30ba.

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-0bdc30ba.

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-0bdc30ba.

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-0bdc30ba.

Para terminar

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-0bdc30ba.

Capítulo 12: El Capstone (Construyendo un Juego)

¿Puede Nanocode construir algo en realidad?

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

Aparte: Esta demo implica muchas llamadas a la API. Si alcanzas los límites de tasa (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:

Aparte: Trabajamos en un subdirectorio para que el `list_files` del agente vea únicamente los archivos del juego, no su propio código fuente, y para que comience con una memoria limpia.

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

Si estás usando el repositorio de código del libro, copia 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
```

Aparte: Los comandos de shell anteriores son para macOS/Linux. En Windows, usa `mkdir snake_game`, `copy nanocode.py snake_game\` y `copy .env snake_game\` en su lugar. Si `pip install pygame` falla en macOS, es posible que necesites `brew install sdl2 sdl2_image sdl2_mixer sdl2_ttf`. En Linux, instala los paquetes de desarrollo de SDL: `sudo apt install libsdl2-dev libsdl2-image-dev libsdl2-mixer-dev libsdl2-ttf-dev`. En Windows, `pip install pygame` incluye todo.

Paso 2: El Arquitecto (Plan Mode)

Inicia el agente. Comenzamos en *plan mode* porque queremos un plano antes de poner 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 usará `write_plan` para crear `PLAN.md`. Léelo. Debería describir la clase Snake, la clase Food y el bucle del juego, todo en un único archivo.

Si el plan parece razonable, pasa al Paso 3.

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 basándose en el contexto que almacenó en `PLAN.md`.

Paso 4: La Prueba de Realidad

Antes de ejecutar el juego, aumenta el tiempo límite. Los 30 segundos predeterminados no son suficientes para jugar una partida completa: el bucle principal de Pygame se bloquea hasta que cierras la ventana. Establece la variable de entorno antes de lanzarlo:

```
export NANOCODE_TIMEOUT=300
```

El Prompt:

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

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

> **Consejo de desarrollo:** 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 bloquea el terminal.

Si se bloquea con un error: Los LLMs son no determinísticos. Es posible que tu agente produzca un error en el primer intento. Si el juego falla con un error como `AttributeError: 'Snake' object has no attribute 'draw'`, no lo corrijas tú mismo. Permite 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 parchearlo y lo ejecutará de nuevo.

Paso 5: El Pivote (Feature Creep)

El juego funciona, pero es feo. La serpiente no es más que cuadrados verdes. Pongamos 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 debería:

1. Usar `search_web` para buscar paletas de colores
2. Usar `read_file` para entender la lógica de renderizado actual
3. Usar `edit_file` para inyectar las nuevas funcionalidades
4. Ejecutar el juego para verificarlo

Qué Sale Mal

Tus resultados serán distintos a los míos: los LLMs son no deterministas. Pero esto es lo que suele ocurrir y en qué debes fijarte.

Fallos habituales 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 primero ejecute `pip install pygame`.
- `AttributeError` en un método que el agente definió pero escribió mal en el punto de llamada. El agente corrige estos errores rápidamente en cuanto ve el traceback.
- Errores de desfase por uno en la detección de colisiones. La serpiente atraviesa las paredes o muere un píxel demasiado pronto. Estos requieren 2-3 iteraciones de editar-ejecutar-corregir.

El arco típico de una sesión:

En mis pruebas, el agente suele conseguir un juego funcional (aunque poco vistoso) en 2-4 iteraciones. La primera escritura produce algo que falla. La segunda o tercera corrección logra que funcione. El paso de incorporación de funcionalidades extra (cambios de color, incrementos 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 entre 15 y 20 rondas. Si usas Claude, presta atención al activador de compactación: alrededor de la ronda 12-15 verás "(Compacting conversation…)" cuando el recuento de tokens se acerque al umbral del 75%. Tras la compactación, el agente pierde algo de detalle sobre las rondas iniciales, pero sigue trabajando. Este es el sistema del Capítulo 9 justificando su valor.

Dónde le cuesta al agente:

El sistema de coordenadas de Pygame y el bucle de eventos son complicados. A veces el agente escribe código que renderiza correctamente, pero no gestiona bien la entrada del teclado, o dibuja la serpiente en el orden incorrecto de modo que la cabeza aparece detrás del cuerpo. Son el tipo de errores que un humano detecta al instante, pero que el agente no puede ver: no tiene feedback visual, solo stdout y stderr. Si el juego se ejecuta sin errores pero se ve mal, tendrás que describir el error visual: "La serpiente se renderiza al revés: la cabeza debería estar al frente."

El objetivo no es la perfección en el primer intento. Es que el agente *converja*: escribir, ejecutar, leer el error, corregir, repetir, usando cada herramienta que construimos a lo largo de once capítulos.

Antes de entregar el `snake.py` definitivo, léelo. El agente escribe código que *funciona*, pero un humano debería igualmente auditarlo en busca de cosas que el bucle de pruebas no puede detectar: números mágicos en el código, casos límite que faltan (¿qué pasa si se redimensiona la ventana?) y si el código está estructurado de una manera que querrías mantener. El agente es un redactor rápido, no un revisor final.

Conclusión

Planificar, implementar, fallar, depurar, corregir, ejecutar de nuevo: cada capítulo demostrando su utilidad.

¿Y a partir de aquí, adónde va?

Epílogo

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

- Integración con Git que haga commits automáticos tras pasar los tests
- Depuración basada en capturas de pantalla para trabajo de frontend (Claude puede leer imágenes)
- Entrada de voz mediante Whisper para que puedas hablar en lugar de escribir
- MCP para conectarte a servicios externos que tu equipo ya utiliza
- Sub-agentes que bifurcan conversaciones y trabajan en subtareas en paralelo

Los agentes de producción como Claude Code, Cursor y Copilot hacen más que esto: respuestas en streaming (ver Apéndice A), ejecución de herramientas en paralelo, tree-sitter parsing, entornos de ejecución aislados, ventanas de contexto que abarcan miles de archivos. La distancia 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 ya sabes qué hay detrás del telón.

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

Apéndice A: Respuestas en Streaming

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-0bdc30ba.

Cómo Funciona el Streaming

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-0bdc30ba.

La Implementación

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-0bdc30ba.

Qué cambió respecto al Capítulo 11

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-0bdc30ba.

La contrapartida

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-0bdc30ba.

Ejecutar el 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-0bdc30ba.

Agradecimientos